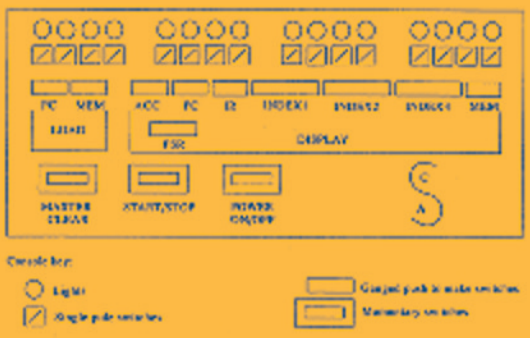
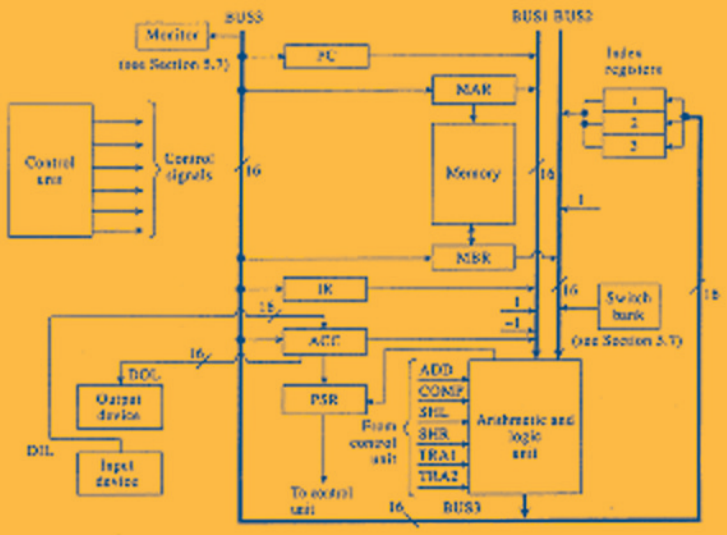
# Computer Design and Architecture

## Third Edition, Revised and Expanded




SAJJAN G. SHIVA

VISIT…

The First and Second Editions were published as
*Computer Design and Architecture* (Little, Brown, 1985; Harper Collins, 1991).

**ISBN: 0-8247-0368-5**
This book is printed on acid-free paper.

**Headquarters**
Marcel Dekker, Inc.
270 Madison Avenue, New York, NY 10016
tel: 212-696-9000; fax: 212-685-4540

**Eastern Hemisphere Distribution**
Marcel Dekker AG
Hutgasse 4, Postfach 812, CH-4001 Basel, Switzerland
tel: 41-61-261-8482; fax: 41-61-261-8896

**World Wide Web**
http://www.dekker.com

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write to Special Sales/Professional Marketing at the headquarters address above.

**Copyright © 2000 by Marcel Dekker, Inc. All Rights Reserved.**

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage and retrieval system, without permission in writing from the publisher.

Current printing (last digit):
10 9 8 7 6 5 4 3 2 1

**PRINTED IN THE UNITED STATES OF AMERICA**

*To*
*My Parents*

# Preface

This book is about the design and architecture of computers. Architecture is the "art or science of building; a method or style of building," according to one dictionary. A computer architect develops the functional and performance specifications for the various blocks of a computer system and defines the interfaces between those blocks, in consultation with hardware and software designers. A computer designer, on the other hand, refines those building-block specifications and implements those blocks with an appropriate mix of hardware, software, and firmware. It is my belief that the capabilities of an architect are greatly enhanced if he or she is exposed to the design aspects of a computer system. Consequently, the backbone of this book is a description of the complete design of a simple but complete hypothetical computer. The book describes the architectural features of existing computer systems as enhancements to the structure of the simple computer.

Books on digital systems architecture fall into four categories: logic design books that cover the hardware logic design in detail but fail to provide the details of computer hardware design; books on computer organization that deal with the computer hardware from a programmer's viewpoint; books on computer hardware design that are suitable for an electrical engineering curriculum; and books on computer system architecture with no detailed treatment of hardware design aspects. I have tried to capture the important attributes of the four categories of books to come up with a comprehensive text on hardware design and architecture. I have included the pertinent software aspects as needed.

The first edition of the book, published in 1985, was a result of my teaching a sequence of computer architecture courses at the senior undergraduate and beginning graduate levels for several years to both computer science and electrical engineering students. The second edition, published in 1991, expanded the book to eleven chapters and included several additional

topics in response to the comments from the users of the first edition. The third edition expands the topical coverage of the second edition and contains more contemporary architectures as examples. The book does not assume prior knowledge of computers, although exposure to programming with high-level and assembly languages makes the reading easier. Exposure to electronics is not required as a prerequisite for understanding this book.

The introductory chapter briefly outlines digital computer system terminology and traces the evolution of computer structures. Traditionally, a discussion of number systems and computer codes is found in an introductory chapter of this kind. I have moved this material to Appendix A, since a majority of students are now familiar with that material much earlier than at the college level. Nevertheless, Appendix A provides sufficient details for those not familiar with that subject.

Chapter 1 covers the analysis and design of combinational logic circuits. Logic-minimization procedures are not part of this chapter, since a mastery of logic minimization is not required for the purposes of this book. Appendix B provides the details of logic minimization. Popular off-the-shelf integrated circuits (ICs) and designing with ICs are introduced in this chapter. Appendix C provides brief descriptions of selected ICs.

Chapter 2 covers the analysis and design of synchronous sequential circuits. Concepts of registers and register-transfer logic are introduced along with primitive register-transfer language (RTL). This RTL is used in the rest of the book, although any other available RTL can be substituted in its place.

Chapter 3 introduces models for commonly used memories and describes various memory devices and organizations with an emphasis on semiconductor memory design.

Chapter 4 provides a programmer's view of a simple hypothetical computer (ASC). ASC organization, instruction set, assembly-language programming, and details of an assembler are provided along with a brief introduction to program loading.

Chapter 5 details the hardware design of ASC, including both hardwired and microprogrammed control units. Although the development of such detailed design is tedious, it is my belief that system designers, architects, and system programmers should go through the steps of such detailed design to better appreciate the inner workings of a digital computer system.

Chapters 1 through 5 form the "design" part of the book while the remaining chapters emphasize the "architecture." These chapters describe the architectural features found in practical computer systems as enhancements to the structure of ASC. Only pertinent details of commercial systems are used as examples rather than complete descriptions of commercial architectures.

Chapter 6 enhances the input/output subsystem of ASC from the programmed input/output structure to the concept of input/output processors, through interrupt processing, direct memory access, and input/output channels. System structures of four popular commercial systems are detailed.

Chapter 7 covers popular data and instruction representations, register structures, and addressing modes.

Chapter 8 details the enhancements to the memory system in terms of cache and virtual memory organizations.

Chapter 9 is devoted to the enhancements to the control unit and covers the topics of pipelining, parallel instruction execution, and reduced instruction set computers (RISC), a popular architectural style of today.

Chapter 10 details the enhancements to the arithmetic/logic unit to cover the concepts of stack-based ALUs, pipelined ALUs, and parallel processing with multiple functional units.

Chapter 11 is a brief introduction to various popular architectures including dataflow architectures, computer networks, and distributed processing. Details of selected system architectures are also provided in this chapter.

Problems are given at the end of each chapter. A solution manual is available from the publisher for the use of instructors. The list of references provided at the end of each chapter may be consulted by readers for further details on topics covered in that chapter.

The following method of study may be followed for a single-semester course in computer architecture:

1. For computer science students with no logic design background: Chapters 1, 2, and 3 in detail, review of Chapter 4, Chapter 5 in detail, selected topics from the remaining chapters.
2. For computer engineering students: Review of Chapters 1 and 2, Chapters 3 through 6 in detail, selected topics from the remaining chapters.

A two-semester course in computer architecture might follow this method of study:

1. Semester 1: Appendix A, Chapter 1, Appendixes B and C, Chapters 2 through 5.
2. Semester 2: Chapters 6 through 11, case studies of selected contemporary systems.

Several of my colleagues and students in my architecture classes at the University of Alabama in Huntsville have contributed immensely to the contents of this book. I thank them. I would also like to thank the users of the first and second editions both in the United States and abroad for their suggestions for improvements that have resulted in this new edition.

Thanks are also due to Russell Dekker and Eric Stannard at Marcel Dekker, Inc., for their encouragement and superb support in the production of the book. I thank my wife, Kalpana, and daughters, Sruti and Sweta, for their love and understanding, and assistance in the preparation of the manuscript.

*Sajjan G. Shiva*

# Contents

*Preface* *v*

**Introduction: Computer System Components** **1**
I.1 Computer System Organization 1
I.2 Computer Evolution 5
I.3 Design vs. Architecture 8
I.4 Overview of the Book 9
References 10

**Chapter 1 Combinational Logic** **11**
1.1 Basic Operations and Terminology 11
1.2 Boolean Algebra (Switching Algebra) 21
1.3 Primitive Hardware Blocks 23
1.4 Functional Analysis of Combinational Circuits 23
1.5 Synthesis of Combinational Circuits 29
1.6 Some Popular Combinational Circuits 35
1.7 Integrated Circuits 47
1.8 Loading and Timing 64
1.9 Summary 68
References 68
Problems 69

**Chapter 2 Synchronous Sequential Circuits** **73**
2.1 Flip-Flops 76
2.2 Timing Characteristics of Flip-Flops 85
2.3 Flip-Flop ICs 91
2.4 Analysis of Synchronous Sequential Circuits 92
2.5 Design of Synchronous Sequential Circuits 102
2.6 Registers 107
2.7 Register Transfer Logic 119

2.8 Register Transfer Schemes 124
2.9 Register Transfer Languages 128
2.10 Designing Sequential Circuits with Integrated Circuits 133
2.11 Summary 136
References 136
Problems 137

**Chapter 3 Memory and Storage 143**
3.1 Types of Memory 144
3.2 Memory System Parameters 152
3.3 Memory Devices and Organizations 156
3.4 Memory System Design Using ICs 173
3.5 Summary 174
References 177
Problems 177

**Chapter 4 A Simple Computer: Organization and Programming 181**
4.1 A Simple Computer 181
4.2 ASC Assembler 197
4.3 Program Loading 209
4.4 Summary 210
References 210
Problems 211

**Chapter 5 A Simple Computer: Hardware Design 215**
5.1 Program Execution 216
5.2 Data, Instruction, and Address Flow 217
5.3 Bus Structure 221
5.4 Arithmetic and Logic Unit 227
5.5 Input/Output 230
5.6 Control Unit 232
5.7 Console 249
5.8 Microprogrammed Control Unit 257
5.9 Summary 268
References 268
Problems 269

**Chapter 6 Input/Output 273**
6.1 General I/O Model 274
6.2 The I/O Function 279
6.3 Interrupts 286
6.4 Direct Memory Access 295

6.5 Bus Architecture 299
6.6 Channels 300
6.7 I/O Processors (IOP) 302
6.8 Serial I/O 306
6.9 Common I/O Devices 310
6.10 Example I/O Structures 313
6.11 Summary 355
References 355
Problems 356

**Chapter 7 Processor and System Structures 359**
7.1 Types of Computer Systems 359
7.2 Operand (Data) Types and Formats 360
7.3 Instruction Set 364
7.4 Registers 372
7.5 Addressing Modes 379
7.6 Example Systems 384
7.7 Summary 420
References 421
Problems 422

**Chapter 8 Memory System Enhancement 425**
8.1 Speed Enhancement 426
8.2 Size Enhancement 441
8.3 Address Extension 447
8.4 Example Systems 450
8.5 Summary 469
References 470
Problems 471

**Chapter 9 Control Unit Enhancement 475**
9.1 Speed Enhancement 476
9.2 Hardwired Control Units 481
9.3 Microprogrammed Control Units 481
9.4 Reduced Instruction Set Computers (RISC) 484
9.5 Example Systems 488
9.6 Summary 519
References 519
Problems 520

**Chapter 10 Arithmetic/Logic Unit Enhancement 523**
10.1 Logical and Fixed-Point Binary Operations 524

10.2 Decimal Arithmetic 535
10.3 Pipelining 536
10.4 ALU with Multiple Functional Units 539
10.5 Example Systems 539
10.6 Summary 568
References 568
Problems 569

**Chapter 11 Advanced Architectures** **571**
11.1 Classes of Architectures 572
11.2 Example Systems 577
11.3 Data-flow Architectures 617
11.4 Computer Networks and Distributed Processing 618
11.5 Summary 622
References 623

*Appendix A Number Systems and Codes* *625*
A.1 Binary System 626
A.2 Octal System 628
A.3 Hexadecimal System 629
A.4 Conversion 629
A.5 Arithmetic 635
A.6 Sign–Magnitude Representation 646
A.7 Complement Number Systems 646
A.8 Codes 651
References 655

*Appendix B Minimization of Boolean Functions* *657*
B.1 Venn Diagrams 657
B.2 Karnaugh Maps 659
B.3 The Quine–McCluskey Procedure 673
B.4 Conclusions 677
References 677

*Appendix C Details of Representative Integrated Circuits* *679*
C.1 Gates, Decoders and Other ICs Useful in Combinational Circuit Design 680
C.2 Flip-Flops, Registers and Other ICs Useful in Sequential Circuit Design 688
C.3 Memory ICs 697
References 708

*Appendix D Stack Implementation* *709*

*Index* *713*

# Introduction: Computer System Components

Recent advances in microelectronic technology have made computers an integral part of our society. Each step in our everyday lives may be influenced by computer technology: we awake to a digital alarm clock's beaming of preselected music at the right time, drive to work in a digital-processor-controlled automobile, work in an extensively automated office, shop for computer-coded grocery items and return to rest in the computer-regulated heating and cooling environment of our homes. It may not be necessary to understand the detailed operating principles of a jet plane or an automobile in order to use and enjoy the benefits of these technical marvels. But a fair understanding of the operating principles, capabilities, and limitations of digital computers *is* necessary, if we would use them in an efficient manner. This book is designed to give such an understanding of the operating principles of digital computers. This chapter will begin by describing the organization of a general-purpose digital computer system and then will briefly trace the evolution of computers.

## I.1 COMPUTER SYSTEM ORGANIZATION

The primary function of a digital computer is to process data input to produce results that can be better used in a specific application environment. For example, consider a digital computer used to control the traffic light at an intersection. The *input* data is the number of cars passing through the intersection during a specified time period, the *processing* consists of the computation of red–yellow–green time periods as a function of the number of cars, and the *output* is the variation of the red–yellow–green time intervals based on the results of processing. In this system, the data input device is a sensor that can detect the passing of a car at the intersection. Traffic lights are the output devices. The electronic device that keeps track of the number of cars and computes the red–yellow–green time periods is the processor.

These physical devices constitute the *hardware* components of the system. The processing hardware is *programmed* to compute the red–yellow–green time periods according to some rule. This rule is the *algorithm* used to solve the particular problem. The algorithm (a logical sequence of steps to solve a problem) is translated into a *program* (a set of instructions) for the processor to follow in solving the problem. Programs are written in a language "understandable" by the processing hardware. The collection of such programs constitutes the *software* component of the computer system.

## Hardware

The traffic-light controller is a very simple special-purpose computer system requiring only a few of the physical hardware components that constitute a general-purpose computer system (see Fig. I.1). The four major hardware

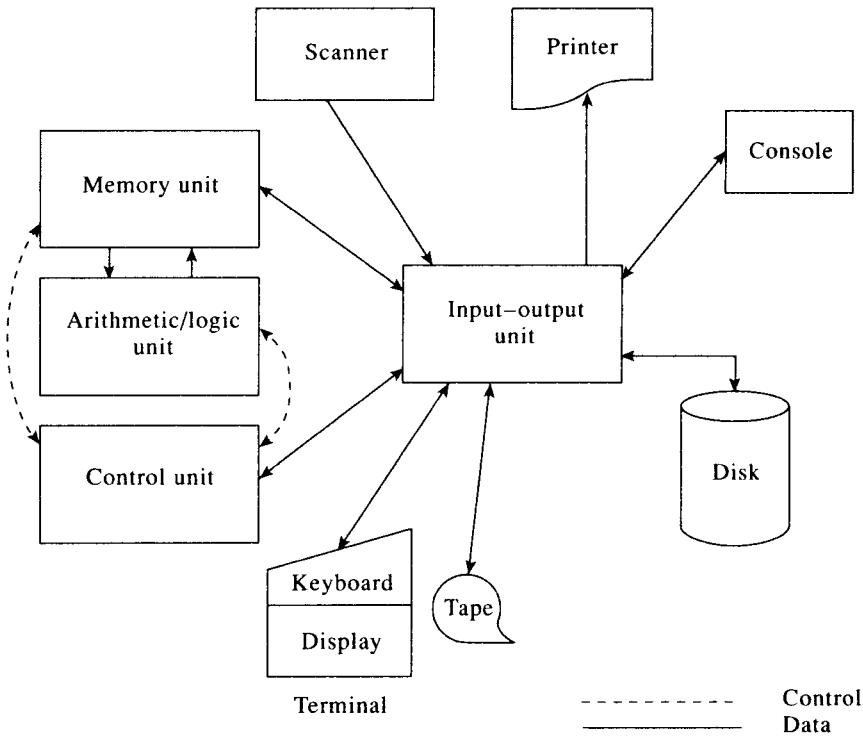



**Figure I.1** A typical computer system

blocks of a general-purpose computer system are its memory unit (MU), arithmetic and logic unit (ALU), input/output unit (IOU), and control unit (CU). Programs and data reside in the memory unit. The arithmetic and logic unit processes the data taken from the memory unit and stores the processed data back in the memory unit. Input/output (I/O) devices input and output data (and programs) into and out of the memory unit. In some systems, I/O devices send and receive data into and from the ALU rather than the MU. The control unit coordinates the activities of the other three units. It retrieves instructions from programs resident in the MU, decodes these instructions, and directs the ALU to perform corresponding processing steps. It also oversees I/O operations.

Several I/O devices are shown in Fig. I.1. A terminal consisting of a keyboard and a video display (i.e., a cathode ray tube – CRT) is the most common I/O device. A printer is the most common output device. Scanners are used to input data from hard-copy sources. Magnetic tapes and disks are used as I/O devices. These devices are also used as memory devices to increase the capacity of the MU. The console is a special-purpose I/O device that permits the system operator to interact with the computer system. In modern-day computer systems, the console is typically a dedicated terminal.

## Software

The hardware components of a computer system are electronic devices in which the basic unit of information is either a 0 or a 1, corresponding to two states of an electronic signal. For instance, in one of the popular hardware technologies a 0 is represented by 0 V while a 1 is represented by +5 V. Programs and data must therefore be expressed using this binary alphabet consisting of 0 and 1. Programs written using only these binary digits are *machine language* programs. At this level of programming, operations such as ADD and SUBTRACT are each represented by a unique pattern of 0s and 1s, and the computer hardware is designed to interpret these sequences. Programming at this level is tedious since the programmer has to work with sequences of 0s and 1s and needs to have very detailed knowledge of the computer structure.

The tedium of machine language programming is partially alleviated by using symbols such as ADD and SUB rather than patterns of 0 and 1 for these operations. Programming at the symbolic level is called *assembly language* programming. An assembly language programmer also is required to have a detailed knowledge of the machine structure, because the operations permitted in the assembly language are primitive and the instruction format

and capabilities depend on the hardware organization of the machine. An *assembler* program is used to translate assembly language programs into machine language.

Use of high-level programming languages such as FORTRAN, COBOL, C and JAVA further reduces the requirement of an intimate knowledge of the machine organization. A *compiler* program is needed to translate a high-level language program into the machine language. A separate compiler is needed for each high-level language used in programming the computer system. Note that the assembler and the compiler are also programs written in one of those languages and can translate an assembly or high-level language program, respectively, into the machine language.

Figure I.2 shows the sequence of operations that occurs once a program is developed. A program written in either the assembly language or a high-level language is called a *source* program. An assembly language source program is translated by the assembler into the machine language program. This machine language program is the *object code*. A compiler converts a high-level language source into object code. The object code ordinarily resides on an intermediate device such as a magnetic disk or tape. A *loader* program loads the object code from the intermediate device into the memory unit. The data required by the program will be either available in the memory or supplied by an input device during the *execution* of the program. The effect of program execution is the production of processed data or results.

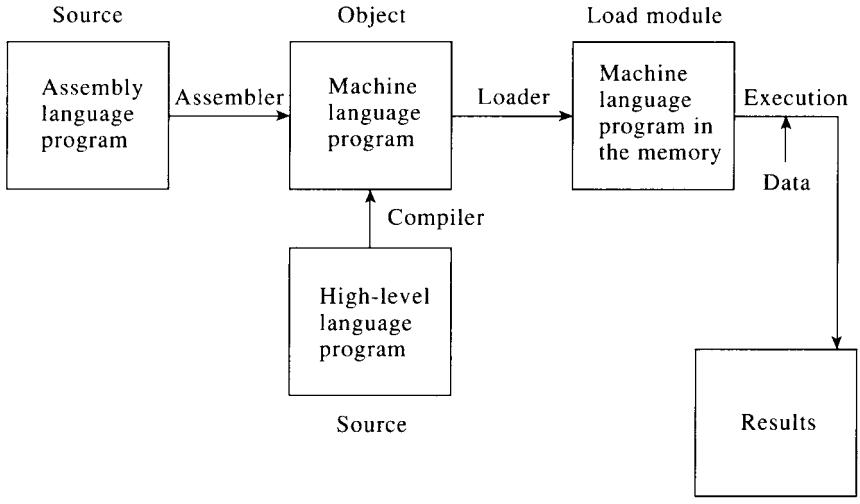



**Figure I.2** Program translation and execution

Operations such as selecting the appropriate compiler for translating the source into object code; loading the object code into the memory unit; and starting, stopping, and accounting for the computer system usage are automatically done by the system. A set of supervisory programs that permit such automatic operation is usually provided by the computer system manufacturer. This set, called the *operating system*, receives the information it needs through a set of command language statements from the user and manages the overall operation of the computer system. Figure I.3 is a simple rendering of the complete hardware–software environment of a general-purpose computer system.

## I.2 COMPUTER EVOLUTION

Man has always been in search of mechanical aids for computation. The development of the abacus around 3000 BC introduced the positional notation of number systems. In seventeenth-century France, Pascal and Leibnitz developed mechanical calculators that were later developed into desk calculators. In 1801, Jacquard used punched cards to instruct his looms in weaving various patterns on cloth.

In 1822, Englishman Charles Babbage developed the difference engine, a mechanical device that carried out a sequence of computations specified by the settings of levers, gears, and cams. Data were entered manually as the

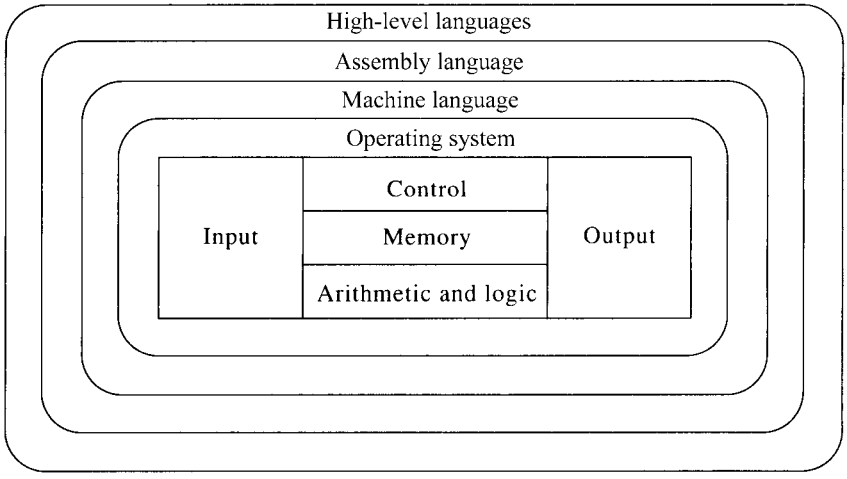



**Figure I.3** Hardware and software components

computations progressed. Around 1820, Babbage proposed the analytical engine, which would use a set of punched cards for program input, another set of cards for data input, and a third set of cards for output of results. The mechanical technology was not sufficiently advanced and the analytical engine was never built; nevertheless, the analytical engine as designed probably was the first computer in the modern sense of the word.

Several unit-record machines to process data on punched cards were developed in the United States in 1880 by Herman Hollerith for census applications. In 1944, Mark I, the first automated computer, was announced. It was an electromechanical device that used punched cards for input and output of data and paper tape for program storage. The desire for faster computations than those Mark I could provide resulted in the development of ENIAC, the first electronic computer built out of vacuum tubes and relays by a team led by Americans Eckert and Mauchly. ENIAC employed the *stored-program concept* in which a sequence of instructions is stored in the memory for use by the machine in processing data. ENIAC had a control board on which the programs were wired. A rewiring of the control board was necessary for each computation sequence.

John von Neumann, a member of the Eckert–Mauchly team, developed EDVAC, the first stored-program computer. At the same time, M. V. Wilkes developed EDSAC, the first operational stored-program machine, which also introduced the concept of primary and secondary memory hierarchy. Von Neumann is credited for developing the stored-program concept, beginning with his 1945 first draft of EDVAC. The structure of EDVAC established the organization of the stored-program computer (von Neumann machine), which contains:

1. An input device through which data and instructions can be entered.
2. A storage into which results can be entered and from which instructions and data can be fetched.
3. An arithmetic unit to process data.
4. A control unit to fetch, interpret, and execute the instructions from the storage.
5. An output device to deliver the results to the user.

All contemporary computers are von Neumann machines, although various alternative architectures are being investigated.

Commercial computer system development has followed development of hardware technology and is usually divided into three generations.

First generation (1954–59) – vacuum tube technology
Second generation (1957–64) – transistor technology

Third generation (mid-1960s) – integrated circuit technology.

We will not elaborate on the architectural details of the various machines developed during the three generations, except for the following brief evolution account.

First-generation machines such as the UNIVAC I and IBM 701, built out of vacuum tubes, were slow and bulky and accommodated a limited number of input–output devices. Magnetic tape was the predominant I/O medium. Data access time was measured in milliseconds.

Second-generation machines (IBM 1401, 7090; RCA 501; CDC 6600; Burroughs 5500; DEC PDP-1) used random-access core memories, transistor technology, multifunctional units, and multiple processing units. Data access time was measured in microseconds. Assembler and high-level languages were developed.

The integrated-circuit technology used in third-generation machines such as the IBM 360, UNIVAC 1108; ILLIAC-IV and CDC STAR-100 contributed to nanosecond data access and processing times. Multiprogramming, array, and pipeline processing concepts came into being.

Computer systems were viewed as general-purpose data processors until the introduction in 1965 of DEC PDP-8, a *minicomputer*. Minicomputers were regarded as dedicated application machines with limited processing capability compared to that of large-scale machines. Since then, several new minicomputers have been introduced and this distinction between the mini and large-scale machines is becoming blurred due to advances in hardware and software technology.

The development of *microprocessors* in the early seventies allowed a significant contribution to the third class of computer systems: microcomputers. Microprocessors are essentially computers on an integrated-circuit (IC) chip that can be used as components to build a dedicated controller or processing system. Advances in IC technology leading to the current very large scale integration (VLSI) era have made microprocessors as powerful as minicomputers of the seventies. VLSI-based systems are called *fourth-generation* systems since their performance is so much higher than that of third generations systems.

Modern computer system architecture exploits the advances in hardware and software technologies to the fullest extent. Due to advances in IC technology that make the hardware much less expensive, the architectural trend is to interconnect several processors to form a high-throughput system.

We are now witnessing the development of *fifth-generation* systems. There is no accepted definition of what a fifth-generation computer is. Fifth-generation development efforts in the U.S.A. involve building supercompu-

ters with very high computational capability, large memory capacity, and flexible multiple-processor architectures. The Japanese fifth-generation activities aimed towards building artificial-intelligence-based machines with very high numeric and symbolic processing capabilities, large memories, and user-friendly natural interfaces.

## I.3 DESIGN vs. ARCHITECTURE

*Architecture* is the art or science of building, a method or style of building. Thus a *computer architect* develops the performance specifications for various components of the computer system and defines the interconnections between them. A *computer designer*, on the other hand, refines these component specifications and implements them using hardware, software, and firmware elements. An architect's capabilities are greatly enhanced if he is also exposed to the design aspects of the computer system.

A computer system can be described at the following levels of detail:

1. Processor-memory-switch (PMS) level, at which an architect views the system. It is simply a description of system components and their interconnections. The components are specified to the block-diagram level.
2. Instruction set level, at which level the function of each instruction is described. The emphasis of this description level is on the *behavior* of the system rather than the hardware *structure* of the system.
3. Register transfer level, at which the hardware structure is more visible compared to previous levels. The hardware elements at this level are registers that retain the data being processed until the current phase of processing is complete.
4. Logic gate level, at which the hardware elements are logic gates and flip-flops. The behavior is now less visible, while the hardware structure predominates.
5. Circuit level, at which the hardware elements are resistors, transistors, capacitors, and diodes.
6. Mask level, at which the silicon structures and their layout that implements the system as an IC are shown.

As one moves from the first level of description towards the last, it is evident that the behavior of the machine is transformed into a hardware–software structure.

A computer architect concentrates on the first two levels described above, while the computer designer takes the system design to the remaining levels.

## I.4 OVERVIEW OF THE BOOK

The basic objective of the book is to provide a comprehensive coverage of both hardware and software aspects of computers with emphasis on hardware design and the architectural tradeoffs required during the design. Therefore, we will stay at the register-transfer and logic gate levels in the first five chapters while dealing with the computer *design* aspects. Chapters 6 through 11 expand on the material from earlier chapters. These chapters are at the computer *architect's* level of detail.

We will provide the detailed design of a hypothetical simple computer to reinforce knowledge of operating principles and discuss the architectural tradeoffs at each stage in the design. No prior exposure to electronics is required to understand the material in this book. We assume a familiarity with binary, octal, and hexadecimal number systems as well as popular computer codes. Appendix A reviews these topics.

Chapters 1 and 2 cover the basic concepts of digital hardware design. The emphasis is on designing logic circuits using off-the-shelf ICs. These chapters, however, are not meant to be a formal introduction to logic design. But it is our belief that the concepts presented here are sufficient preparation for the latter chapters in the book. Although logic minimization theory is important, a mastery of it is not required for the purposes of this book. For reference, a summary of logic minimization is provided in Appendix B.

Chapter 3 outlines memory system organization and discusses popular memory devices. A simple hypothetical computer (ASC) is introduced in Chapter 4. The description of ASC in this chapter is written from a programmer's point of view and includes both the hardware and software components of the machine. A detailed logic design of ASC is provided in Chapter 5.

Chapters 6 through 10 expand the concepts discussed in earlier chapters relative to the architectures of several commercially available machines. Chapter 6 details input/output mechanisms. Chapter 7 introduces the architectural parameters considered in developing processor and system structures. Chapter 8 expands the discussion of memory system design of Chapter 3 to cache and virutal memory schemes. Chapter 9 examines the enhancements possible for the control unit designed in Chapter 5, in light of commercially available architectures, and describes microprogramming

further. Chapter 10 expands the arithmetic-logic unit design of Chapter 5. Chapter 11 is a brief introduction to recent trends in architectures and covers multiprocessor systems.

## REFERENCES

Burks, A. W., Goldstine, H. H. and von Neumann, J. "Preliminary discussion of the logical design of electrical computing instrument," *U.S. Army Ordnance Department report*, 1946.

Culler, D. E., Singh, J. and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA: Morgan Kaufmann, 1998.

Goldstine, H. H. *The Computer from Pascal to von Neumann*, Princeton, NJ: Princeton University Press, 1972.

Hill, M. D., Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.

Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.

# 1
# Combinational Logic

Each hardware component of a computer system is built of several logic circuits. A logic circuit is an interconnection of several primitive logic devices to perform a desired function. It has one or more inputs and one or more outputs. This chapter introduces some logic devices that are used in building one type of logic circuit called a *combinational circuit*. Each output of a combinational circuit is a function of all the inputs to the circuit. Further, the outputs at any time are each a function of inputs at that particular time and so the circuit does not have a memory. A circuit with a memory is called a *sequential circuit*. The output of a sequential circuit at any time is a function of not only the inputs at the time but also the *state* of the circuit at that time. The state of the circuit is dependent on what has happened to the circuit prior to that time and hence the state is also a function of the previous inputs and states. This chapter is an introduction to the analysis and design of combinational circuits. Details of sequential circuit analysis and design are given in Chapter 2.

## 1.1 BASIC OPERATIONS AND TERMINOLOGY

---

**Example 1.1** Consider the addition of two bits:

0 plus 0 = 0
0 plus 1 = 1
1 plus 0 = 1
1 plus 1 = 10 (i.e., a SUM of 0 and a CARRY of 1)

The addition of two single-bit numbers produces a SUM bit and a CARRY bit. The above operations can be arranged into the following table to separate the two resulting bits SUM and CARRY:

| | | *A* plus *B* | |
|---|---|---|---|
| *A* | *B* | SUM | CARRY |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

*A* and *B* are the two operands. Each can take a value of either 0 or 1. The first two columns show the four combinations of values possible for the two operands *A* and *B* and the last two columns represent the sum of the two operands represented as a SUM and a CARRY bit.

Note that the CARRY is 1 only when *A* is 1 *and B* is 1, while the SUM bit is 1 when one of the following two conditions is satisfied: *A* is 0 AND *B* is 1; *A* is 1 and *B* is 0. That is, SUM is 1 if (*A* is 0 AND *B* is 1) OR (*A* is 1 AND *B* is 0). Let us say $A'$ (pronounced "not *A*") represents the opposite condition of *A*; that is, *A* is 0 if $A'$ is 1 and vice versa, and similarly, $B'$ represents the opposite condition of *B*. Then we can say SUM is 1 if ($A'$ is 1 and *B* is 1) OR (*A* is 1 AND $B'$ is 1). A shorthand notation for this is as follows:

$$\begin{aligned} \text{SUM} &= (A' \cdot B) + (A \cdot B') \\ \text{CARRY} &= A \cdot B \end{aligned} \tag{1.1}$$

where

"+" represents the OR operation (not arithmetic addition),
"·" represents the AND operation, and
"′" represents the NOT operation (complement operation).
"−" is also used to represent the NOT operation, as in $\bar{A}$, $\bar{B}$, etc

---

The definition of AND, OR, and NOT operations are shown in Figure 1.1. The right side of Eqs (1.1) are Boolean *expressions*. *A* and *B* are Boolean *variables*. Possible *values* for the Boolean variables in the above example are 0 and 1 (could also be True or False). An expression is formed by combining variables with operations. The value of SUM depends on the values of *A* and *B*. That is, SUM is a *function* of *A* and *B* and so is CARRY.

AND

| $A$ | $B$ | $A \cdot B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

"0" if at least $A$ or $B$ is "0."

"1" only if both $A$ and $B$ are "1".

OR

| $A$ | $B$ | $A + B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

"0" only if both $A$ and $B$ are "0."

"1" if at least $A$ or $B$ is "1."

NOT

| $A$ | $A'$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

$A'$ is the COMPLEMENT of $A$.

**Figure 1.1** Basic operations: AND, OR, and NOT

---

**Example 1.2** Consider the following statement: Subtract if and only if an add instruction is given and the signs are different or a subtract instruction is given and the signs are alike.

Let

- $S$ represent the "subtract" action,
- $A$ represent "add instruction given" condition,
- $B$ represent "signs are different" condition, and
- $C$ represent "subtract instruction given" condition.

Then, the above statement can be expressed as

$$S = (A \cdot B) + (C \cdot B') \tag{1.2}$$

Usually, the "$\cdot$" and "( )" are removed from expressions when there is no ambiguity. Thus, the above function can be written as

$$S = AB + CB' \tag{1.3}$$

---

### 1.1.1 Evaluation of Expressions

Knowing the value of each of the component variables of an expression, we can find the value of the expression itself. The hierarchy of operations is important in the evaluation of expressions. We always perform NOT operations first, followed by AND and lastly OR, in the absence of parentheses. If there are parentheses, the expressions within the parentheses are evaluated first, observing the above hierarchy of operations, and then the remaining expression is evaluated. That is,

> Perform
>
> Parenthesis grouping (if any). Then,
> NOT operation first,
> AND operation next,
> OR operation last,
>
> while evaluating an expression.

The following examples illustrate expression evaluation. The sequential order in which the operations are performed is shown by the numbers below each operation.

---

**Example 1.3** Evaluate $AB' + BC'D$.

$$\begin{array}{lll} A \cdot B' + B \cdot C' \cdot D & & \text{Insert ``}\cdot\text{''} \\ \quad 1 \quad\quad 2\_\_\_\_\_ & & \text{Scan 1 for NOT operations.} \\ 3 \quad\quad 4 \quad 5\_\_\_\_\_ & & \text{Scan 2 for AND operations.} \\ \quad 6\_\_\_\_\_ & & \text{Scan 3 for OR operations.} \end{array}$$

---

---

**Example 1.4** Evaluate A $(B + C'D) + AB' + C'D'$.

$$\begin{array}{lll} A \cdot (B + C' \cdot D) + A \cdot B' + C' \cdot D' & & \text{Insert ``}\cdot\text{''} \\ \quad 1 & \left.\begin{array}{l}\text{NOT}\\ \text{AND}\\ \text{OR}\end{array}\right\} & \text{Within parentheses} \\ \quad\quad 2 & & \\ 3 & & \\ \quad\quad\quad\quad 4 \quad 5 \quad 6 & & \text{NOT} \\ 7 \quad\quad\quad 8 \quad\quad 9 & & \text{AND} \\ \quad\quad 10 \quad\quad 11 & & \text{OR} \end{array}$$

---

**Example 1.5** Evaluate the function $Z = AB'C + (A'B)(B + C')$, given $A = 0$, $B = 1$, $C = 1$.

| | |
|---|---|
| $Z = (A \cdot B' \cdot C) + (A' \cdot B) \cdot (B + C')$ | Insert "$\cdot$" |
| $= (0 \cdot 1' \cdot 1) + (0' \cdot 1) \cdot (1 + 1')$ | Substitute values. |
| $= (0 \cdot 0 \cdot 1) + (1 \cdot 1) \cdot (1 + 0)$ | Evaluate NOT. |
| $= (0) + (1) \cdot (1)$ | Evaluate parenthetical expressions. |
| $= 0 + 1$ | AND operation. |
| $= 1$ | OR operation (value of $Z$ is 1). |

### 1.1.2 Truth Tables

Figure 1.1 shows truth tables for the three primitive operations AND, OR, and NOT. A truth table indicates the value of a function for all possible combinations of the values of the variables of which it is a function. There will be one column in the truth table corresponding to each variable and one column for the value of the function. Since each variable can take either of the two values (0 or 1), the number of combinations of values multiplies as the number of component variables increases. For instance, if there are two variables, there will be $2 \times 2 = 4$ combinations of values and hence four rows in a truth table. In general, there will be $2^N$ rows in a truth table for a function with $N$ component variables. If the expression on the right-hand side of a function is complex, the truth table can be developed in several steps. The following example illustrates the development of a truth table.

**Example 1.6** Draw a truth table for $Z = AB' + A'C + A'B'C$.

There are three component variables, $A$, $B$, and $C$. Hence there will be $2^3$ or 8 combinations of values of $A$, $B$, and $C$. The 8 combinations are shown on the left-hand side of the truth table in Figure 1.2. These combinations are generated by changing the value of $C$ from 0 to 1 and from 1 to 0 as we move down from row to row, while changing the value of $B$ once every two (i.e., $2^1$) rows and changing the value for $A$ once every four (i.e., $2^2$) rows. These combinations are thus in a numerically increasing order in the binary number system, starting with $(000)_2$ or $(0)_{10}$ to $(111)_2$ or $(7)_{10}$, where the subscripts denote the base of the number system.

In general, if there are $N$ component variables, there will be $2^N$ combinations of values ranging in their numerical value from 0 to $2^N - 1$.

| $A$ | $B$ | $C$ | $A'$ | $B'$ | $AB'$ | $A'C$ | $A'B'C$ | $Z$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 1.2** Truth table for $Z = AB' + A'C + A'B'C$

To evaluate $Z$ in the example function, knowing the values for $A$, $B$, and $C$ at each row of the truth table in Fig. 1.2, values of $A'$ and $B'$ are first generated; values for $(AB')$, $(A'C)$ and $(A'B'C)$ are then generated by ANDing the values in appropriate columns at each row; and finally the value of $Z$ is found by ORing the values in the last three columns at each row. Note that evaluating $A'B'C$ corresponds to ANDing $A'$ and $B'$, values followed by ANDing the value of $C$. Similarly, if more than two values are to be ORed, they are ORed two at a time. The columns corresponding to $A'$, $B'$, $(AB')$, $(A'C)$, and $(A'B'C)$ are not usually shown in the final truth table.

### 1.1.3 Functions and Their Representation

There are two *constants* in the logic alphabet: 0 and 1 (true or false). A *variable* such as $A$, $B$, $X$, or $Y$ can take the value of either 1 or 0 at any time. There are three basic operations: AND, OR, and NOT. When several variables are ANDed together, we get a *product* term (conjunction).

**Example 1.7** $AB'C, AB'XY'Z'$.

When several variables are ORed together, we get a *sum* term (disjunction).

**Example 1.8** $(A + B + C'), (X + Y'), (P + Q' + R')$.

Each occurrence of a variable either in *true* form or in *complemented* (inversed, NOT) form is a *literal*. For example, the product term $XY'Z$ has three literals; the sum term $(A' + B' + C + D)$ has four literals.

A product term (sum term) $X$ is *included* in another product term (sum term) $Y$ if $Y$ has each literal that is in $X$.

---

**Example 1.9** $XY'$ is included in $XY'$. $XY'Z$ is included in $XY'ZW$. $(X' + Y)$ is included in $(X' + Y + W)$. $XY'$ is not included in $XY$. (Why?)

---

If the value of the variable $Q$ is dependent on the value of several variables (say $A$, $B$, $C$) – that is, $Q$ is a function of $A$, $B$, $C$ – then $Q$ can be expressed as a sum of several product terms in $A$, $B$, $C$.

---

**Example 1.10** $Q = AB' + A'C + B'C$ is the sum of products (SOP) form.

---

If none of the product terms is included in the other product terms, we get a *normal sum of products form*.

---

**Example 1.11** $Q = AB + AC$, $Q = X + Y$, and $P = AB'C + A'CD + AC'D'$ are in normal SOP form.

---

Similarly, we can define a *normal product of sums form*.

---

**Example 1.12** $P = (X + Y') \cdot (X' + Y' + Z')$ and $Q = (A + B') \cdot (A' + B + C') \cdot (A + B + C)$ are in normal product of sums (POS) form.

---

A truth table can be used to derive the function in SOP or POS forms, as detailed below.

**Example 1.13** Consider the following truth table for $Q$, a function of $A$, $B$, and C:

| $A$ | $B$ | $C$ | $Q$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(NOTE: This is the same as Figure 1.2)

From the truth table it can be seen that $Q$ is 1 when $A = 0$ and $B = 0$ and $C = 1$. That is, $Q$ is 1 when $A' = 1$ and $B' = 1$ and $C = 1$, which means $Q$ is 1 when $(A' \cdot B' \cdot C)$ is 1. Similarly, corresponding to the other three ones in the $Q$ column of the table, $Q$ is 1 when $(A'BC)$ is 1 or $(AB'C')$ is 1 or $(AB'C)$ is 1. This argument leads to the following representation for $Q$:

$$Q = A'B'C + A'BC + AB'C' + AB'C$$

which is the normal SOP form.

In general, to derive an SOP form from the truth table, we can use the following procedure:

1. Generate a product term corresponding to each row where the value of the function is 1.
2. In each product term, consider the individual variables uncomplemented if the value of the variable in that row is 1 and complemented if the value of the variable in that row is 0.

The POS form for the function can be derived from the truth table by a similar procedure:

1. Generate a sum term corresponding to each row where the value of the function is 0.
2. In each sum term, consider the individual variables complemented if the value of the variable in that row is 1 and uncomplemented if the value of the variable in that row is 0.

$Q = (A + B + C) \cdot (A + B' + C) \cdot (A' + B' + C) \cdot (A' + B' + C')$ is the POS form for $Q$, in Example 1.13.

The SOP form is easy and natural to work with compared to the POS form. The POS form tends to be confusing to the beginner since it does not correspond to the algebraic notation that we are used to.

---

**Example 1.14** Derivation of SOP and POS forms of representation for another three-variable function, $P$, is shown here:

| | $A$ | $B$ | $C$ | $P$ | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | $\leftarrow A'B'C'$ |
| 1 | 0 | 0 | 1 | 0 | $\leftarrow (A + B + C')$ |
| 2 | 0 | 1 | 0 | 0 | $\leftarrow (A + B' + C)$ |
| 3 | 0 | 1 | 1 | 0 | $\leftarrow (A + B' + C')$ |
| 4 | 1 | 0 | 0 | 1 | $\leftarrow AB'C'$ |
| 5 | 1 | 0 | 1 | 1 | $\leftarrow AB'C$ |
| 6 | 1 | 1 | 0 | 0 | $\leftarrow (A' + B' + C)$ |
| 7 | 1 | 1 | 1 | 0 | $\leftarrow (A' + B' + C')$ |

SOP form: $P = A'B'C' + AB'C' + AB'C$

POS form: $P = (A + B + C') \cdot (A + B' + C) \cdot (A + B' + C') \cdot (A' + B' + C) \cdot (A' + B' + C')$

---

### 1.1.4 Canonical Forms

The SOP and POS forms of the functions derived from a truth table by the above procedures are *canonical forms*. In a canonical SOP form each component variable appears in either complemented or uncomplemented form in each product term.

---

**Example 1.15** If $Q$ is a function of $A$, $B$, and $C$, then $Q = A'B'C + AB'C' + A'B'C'$ is a canonical SOP form, while $Q = A'B + AB'C + A'C'$ is not, because in the first and last product terms, all three variables are not present.

---

A canonical POS form is similarly defined. A canonical product term is also called a *minterm*, while a canonical sum term is called a *maxterm*.

Hence, functions can be represented either in *sum of minterm* or in *product of maxterm* formats.

---

**Example 1.16** From the truth table of Example 1.14:

$$P(A, B, C) = A'B'C' + AB'C' + AB'C$$

0 0 0 1 0 0 1 0 1 ← Input combinations (0 for a complemented variable; 1 for an uncomplemented variable)

0 4 5 ← Decimal values

$= \Sigma m(0, 4, 5)$ ← Minterm list form

The minterm list form is a compact representation for the canonical SOP form.

$$P(A, B, C) = (A + B + C') \cdot (A + B' + C) \cdot (A + B' + C') \cdot (A' + B' + C) \cdot (A' + B' + C')$$

0 0 1 0 1 0 0 1 1 1 1 0 1 1 1

Input combinations (1 for a complemented variable and 0 for an uncomplemented variable)

1 2 3 6 7 ← Decimal values

$= \Pi M(1, 2, 3, 6, 7)$ ← Maxterm list form

The maxterm list form is a compact representation for the canonical POS form. Knowing one form, the other can be derived as shown by the following example.

---

**Example 1.17** Given $Q(A, B, C, D) = \Sigma m(0, 1, 7, 8, 10, 11, 12, 15)$.

$Q$ is a four-variable function. Hence, there will be $2^4$ or 16 combinations of input values whose decimal values range from 0 to 15. There are eight minterms. Hence, there should be $16 - 8 = 8$ maxterms: that is,

$$Q(A, B, C, D) = \Pi M(2, 3, 4, 5, 6, 9, 13, 14).$$

Also, note that the complement of $Q$ is represented as

$$Q'(A, B, C, D) = \Sigma m(2, 3, 4, 5, 6, 9, 13, 14)$$
$$= \Pi M(0, 1, 7, 8, 10, 11, 12, 15).$$

---

Note that for an $n$ variable function,

$$\text{(Number of minterms)} + \text{(number of maxterms)} = 2^n \tag{1.4}$$

## 1.2 BOOLEAN ALGEBRA (SWITCHING ALGEBRA)

In 1854, George Boole introduced a symbolic notation to deal with symbolic statements that take a binary value of either *true* or *false*. This symbolic notation was adopted by Claude Shannon to analyze logic functions and has since come to be known as Boolean algebra or switching algebra. The definitions, theorems, and postulates of this algebra are described below.

**Definition** A Boolean algebra is a closed algebraic system containing a set $K$ of two or more elements and two binary operators "$+$" (OR) and "$\cdot$" (AND); that is, for every $X$ and $Y$ in set $K$, $X \cdot Y$ belongs to $K$, and $X + Y$ belongs to $K$. In addition, the following postulates must be satisfied.

**Postulates**

| | |
|---|---|
| P1 Existence of 1 and 0 | (a) $X + 0 = X$<br>(b) $X \cdot 1 = X$ |
| P2 Commutativity | (a) $X + Y = Y + X$<br>(b) $X \cdot Y = Y \cdot X$ |
| P3 Associativity | (a) $X + (Y + Z) = (X + Y) + Z$<br>(b) $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$ |
| P4 Distributivity | (a) $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$<br>(b) $X \cdot (Y + Z) = X \cdot Y + X \cdot Z$ |
| P5 Complement | (a) $X + X' = 1$ ($X'$ is the complement of $X$)<br>(b) $X \cdot X' = 0$ |

**Definition** Two expressions are said to be *equivalent* if one can be replaced by the other.

**Definition** The "dual" of an expression is obtained by replacing each "$+$" in the expression by "$\cdot$", each "$\cdot$" by "$+$", each 1 by 0, and each 0 by 1.

The *principle of duality* states that if an equation is valid in a Boolean algebra, its dual is also valid.

---

**Example 1.18** Given $X + YZ = (X + Y) \cdot (X + Z)$, its dual is $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$.

---

Note that part (b) of each of the postulates is the dual of the corresponding part (a).

**Theorems** The following theorems are useful in manipulating Boolean functions. They are traditionally used for converting Boolean functions from one form to another, deriving canonical forms, and minimizing (reducing the complexity of) Boolean functions. These theorems can be proven by drawing truth tables for both sides to see if the left-hand side has the same values as the right-hand side, for each possible combination of component variable values.

| | |
|---|---|
| T1 Idempotency | (a) $X + X = X$<br>(b) $X \cdot X = X$ |
| T2 Properties of 1 and 0 | (a) $X + 1 = 1$<br>(b) $X \cdot 0 = 0$ |
| T3 Absorption | (a) $X + XY = X$<br>(b) $X \cdot (X + Y) = X$ |
| T4 Absorption | (a) $X + X'Y = X + Y$<br>(b) $X \cdot (X' + Y) = X \cdot Y$ |
| T5 DeMorgan's law | (a) $(X + Y)' = X' \cdot Y'$<br>(b) $(X \cdot Y)' = X' + Y'$ |
| T6 Consensus | (a) $XY + X'Z + YZ = XY + X'Z$<br>(b) $(X + Y) \cdot (X' + Z) \cdot (Y + Z) =$<br>$(X + Y) \cdot (X' + Z)$ |

We can summarize some important properties thus:

$$X + 0 = X \quad X + 1 = 1$$
$$X \cdot 0 = 0 \quad X \cdot 1 = X$$
$$0' = 1, 1' = 0, X'' = (X')' = X$$

Algebraic proofs for the above theorems can be found in any of the references listed at the end of this chapter. Appendix B deals with the applications of Boolean algebra for minimization of logic functions. These minimization techniques are useful in reducing the complexity of logic circuits. We will not emphasize minimization in this book.

## 1.3 PRIMITIVE HARDWARE BLOCKS

A logic circuit is the physical implementation of a Boolean function. The primitive Boolean operations AND, OR, and NOT are implemented by electronic components known as *gates*. The Boolean constants 0 and 1 are implemented as two unique voltage levels or current levels. A gate receives these logic values on its inputs and produces a logic value that is a function of its inputs on its output. Each gate has one or more inputs and an output. The operation of a gate can be described by a truth table. The truth tables and standard symbols used to represent the three primitive gates are shown in Fig. 1.3.

The NOT gate will always have one input and one output. Only two inputs are shown for the other gates in Fig. 1.3 for convenience. The maximum number of inputs allowed on the gate is limited by the electronic technology used to build the gate. The number of inputs on the gate is termed its *fan-in*. We will assume that there is no restriction on the fan-in. A four-input AND gate is shown below.

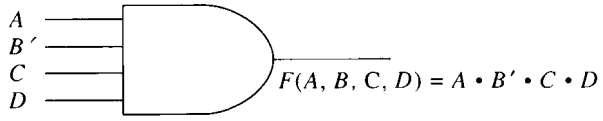



Figure 1.3 shows three other popular gates. The utility of these gates will be discussed later in this chapter.

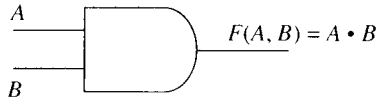
## 1.4 FUNCTIONAL ANALYSIS OF COMBINATIONAL CIRCUITS

The *functional analysis* of a combinational circuit is the process of determining the relations of its outputs to its inputs. These relations are expressed as either a set of Boolean functions, one for each output, or as a truth table for the circuit. The functional analysis is usually performed to verify the stated function of a combinational circuit. If the function of the circuit is not known, the analysis determines it. Two other types of analysis are commonly performed on logic circuits. They are *loading* and *timing* analyses. We will discuss functional analysis in this section and describe the other types of analyses in Section 1.8.

Consider the combinational circuit with $n$ input variables and $m$ outputs shown in Fig. 1.4 (a). Since there are $n$ inputs, there are $2^n$ combinations of input values. For each of these input combination, there is a unique combination of output values. A truth table for this circuit will have $2^n$ rows

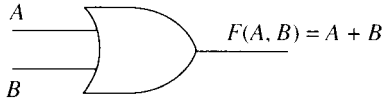
Operation Symbol Truth table

AND



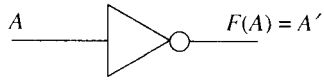

| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR



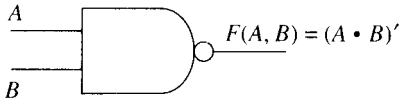

| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

NOT



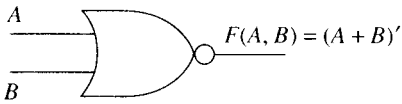

| A | F(A) |
|---|---|
| 0 | 1 |
| 1 | 0 |

NAND



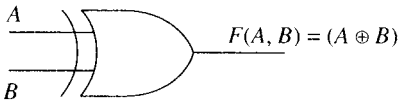

| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NOR




| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

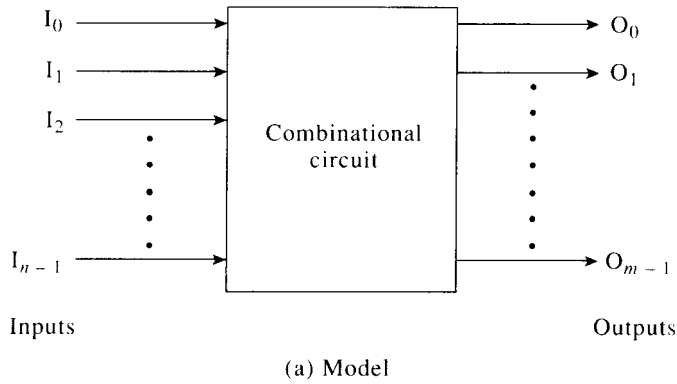
Exclusive OR



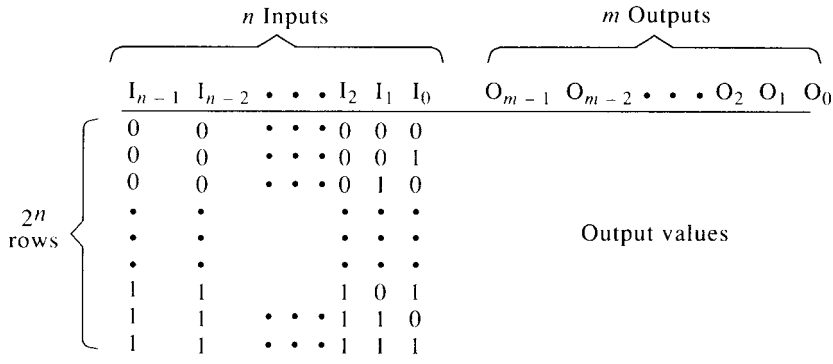

$F(A, B) = A \oplus B = A'B + AB'$

| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Figure 1.3** Gates and truth tables




(a) Model




(b) Truth table

**Figure 1.4** Combinational circuit

and $(n + m)$ columns, as shown in (b). We will need $m$ Boolean functions to describe this circuit. We will demonstrate the analysis procedure in the following example.

---

**Example 1.19** Consider the circuit shown in Fig. 1.5. The three input variables to the circuit are $X$, $Y$, and $Z$; the two outputs are $P$ and $Q$.

In order to derive the outputs $P$ and $Q$ as functions of $X$, $Y$, and $Z$, we can trace through the signals from inputs to the outputs. To facilitate such a tracing, we have labeled the outputs of all the gates in the circuit with the arbitrary symbols $R_1$, $R_2$, $R_3$, and $R_4$. Note that $R_1$ and $R_2$ are functions of only input variables; $R_3$ and $R_4$ are functions of $R_1$, $R_2$, and the input

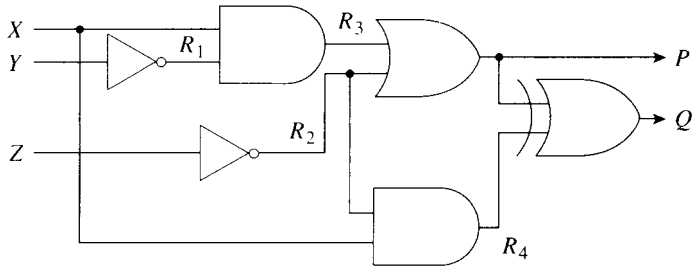



**Figure 1.5** Circuit with three input variables

variables. $P$ is a function of $R_2$ and $R_3$, and $Q$ is a function of $P$ and $R_4$. Tracing through the circuit, we note that:

$$R_1 = Y' \qquad \text{and } R_2 = Z'$$

$$R_3 = X \cdot R_1$$
$$= X \cdot Y' \qquad = XY'$$

$$R_4 = R_2 \cdot X$$
$$= Z' \cdot X \qquad = XZ'$$

$$P = R_3 + R_2$$
$$= XY' + Z'$$

$$Q = P \oplus R_4$$
$$= (XY' + Z') \oplus XZ'$$

We can transform the function for $Q$ into a SOP form using theorems and postulates of Boolean algebra as shown below. The theorems and postulates used are identified with T and P respectively along with their numbers.

$$Q = \underset{\text{T5a}}{(XY' + Z')'} \cdot (XZ') + (XY' + Z') \cdot \underset{\text{T5b}}{(XZ')'}$$

$$= \underset{\text{T5b Involution}}{(XY')'} \cdot (Z')' \cdot (XZ') + (XY' + Z') \cdot \underset{\text{Involution}}{(X' + Z'')}$$

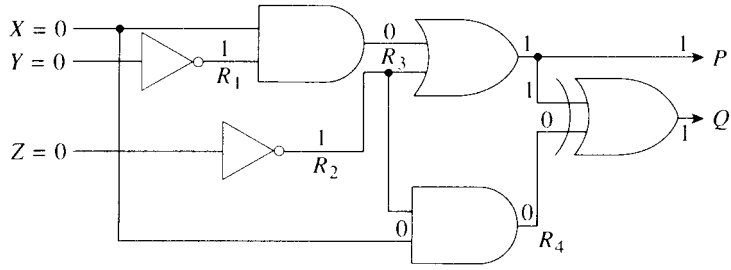
$$= \underset{\text{Involution}}{(X' + Y'')} \cdot Z \cdot (XZ') + \underset{\text{P4b}}{(XY' + Z' \cdot (X' + Z)}$$

$$
\begin{aligned}
&= (X' + Y) \cdot Z \cdot Z'X + (XY' + Z')X' + (XY' + Z')Z \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{p4b} \qquad\qquad \text{p4b} \\
&= \quad 0 \qquad\qquad\qquad + \; XY'X' + Z'X' + XY'Z + ZZ' \\
&\qquad\qquad\qquad\qquad\qquad\quad 0 \qquad\qquad\qquad\quad 0 \\
&= \quad X'Z' + XY'Z
\end{aligned}
$$

Thus,

$$
\begin{aligned}
P &= XY' + Z' \\
Q &= X'Z' + XY'Z.
\end{aligned}
$$

Figure 1.6 shows the derivation of the truth table for the above circuit. The truth table can be drawn from the above functions for $P$ and $Q$ or by tracing through the circuit. Since there are three inputs, the truth table will have eight rows. The eight combinations of input values are shown in Fig. 1.6(b). We can now impose each combination of values on the inputs of the circuit and note the output values, tracing through the circuit. Figure. 1.6(a)




(a) Circuit with 000 input condition

| X Y Z | $R_1$ | $R_2$ | $R_3$ | $R_4$ | P | Q |
|---|---|---|---|---|---|---|
| 0 0 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 0 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 1 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 1 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 0 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 0 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 1 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 1 1 | 0 | 0 | 0 | 0 | 0 | 0 |

$R_1 = Y'$
$R_2 = Z'$
$R_3 = X \cdot R_1$
$R_4 = X \cdot R_2$
$P = R_2 + R_3$
$Q = P \oplus R_4$

(b) Truth table

**Figure 1.6** Derivation of truth table

shows the condition of the circuit corresponding to the input condition $X = 0$, $Y = 0$, and $Z = 0$. Tracing through the circuit we note that $R_1 = 1$, $R_2 = 1$, $R_3 = 0$, $R_4 = 0$, $P = 1$, and $Q = 1$. We repeat this process with the other seven input combinations to derive the complete truth table shown in (b).

We have shown the columns corresponding to intermediate outputs $R_1$, $R_2$, $R_3$, and $R_4$ in the above truth table for convenience only. These columns are not usually retained in the final truth table.

---

We summarize the functional analysis procedures below.
To obtain the output functions from a logic diagram:

1. Label outputs of all gates in the circuit with arbitrary variable names.
2. Express the outputs of the first level of gates (i.e., the gates whose inputs are circuit input variables) as functions of their inputs.
3. Express the outputs of the next level of gates as functions of their inputs (which are circuit inputs and outputs from the previous level of gates).
4. Continue the process of step 3 until the circuit outputs are obtained.
5. Substitute the functions corresponding to each intermediate variable into the output functions, to eliminate the intermediate variables from the functions.
6. Simplify the output functions (if possible) using the properties of Boolean algebra or by the methods described in Appendix B.

To obtain the truth table from the circuit diagram:

1. Determine the number of inputs $n$ in the circuit. List the binary numbers from 0 through $(2^n - 1)$ forming the input portion of the truth table with $2^n$ rows.
2. Label all the outputs of all the gates in the circuit with arbitrary symbols.
3. Determine the outputs of the first level of gates for each input combination. Each first-level output forms a column in the truth table.
4. Using the combination of input values and the values of intermediate outputs already determined, derive the values for the outputs of the next level of gates.
5. Continue the process in step 4 until the circuit outputs are reached.

## 1.5 SYNTHESIS OF COMBINATIONAL CIRCUITS

*Synthesis* is the process of transforming the word statement of the function to be performed into a logic circuit. The word statement is first converted into a truth table. This step requires the identification of the input variables and the output values corresponding to each combination of input values. Each output can then be expressed as a Boolean function of input variables. The functions are then transformed into logic circuits. In addition to "synthesis," several other terms are used in the literature to denote this transformation. We say that we "realize" the function by the circuit; we "implement" the function using the logic circuit; we "build" the circuit from the function; or simply "design" the circuit. In this book we use all these terms interchangeably as needed.

Four types of circuit implementations are popular. AND–OR and NAND–NAND implementations can be generated directly from the SOP form of the function: OR–AND and NOR–NOR implementations evolve directly from the POS form. We will illustrate these implementations using the following example.

---

**Example 1.20** Build a circuit to implement the function $P$ shown in the following truth table:

| $X$ | $Y$ | $Z$ | $P$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $(X + Y + Z)$ |
| 0 | 0 | 1 | 0 | $(X + Y + Z')$ |
| 0 | 1 | 0 | 1 | $X'YZ'$ |
| 0 | 1 | 1 | 1 | $X'YZ$ |
| 1 | 0 | 0 | 0 | $(X' + Y + Z)$ |
| 1 | 0 | 1 | 1 | $XY'Z$ |
| 1 | 1 | 0 | 0 | $(X' + Y' + Z)$ |
| 1 | 1 | 1 | 1 | $XYZ$ |
| Inputs | | | Output | |

---

### 1.5.1 AND–OR Circuits

Let us express $P$ from the above example in SOP form:

$$P = X' \cdot Y \cdot Z' + X' \cdot Y \cdot Z + X \cdot Y' \cdot Z + X \cdot Y \cdot Z$$

$P(X, Y, Z)$ is the sum of four product terms, so we use an OR gate with four inputs to generate $P$ [see Fig. 1.7(a)]. Each of the inputs to this OR gate is a product of three variables. Hence we use four AND gates, each realizing a product term. The outputs of these AND gates are connected to the four inputs of the OR gate as shown in Fig. 1.7(b).

A complemented variable can be generated using a NOT gate. Figure 1.7(c) shows the circuit needed to generate $X'$, $Y'$, and $Z'$. The final task in building the circuit is to connect these complemented and uncomplemented signals to appropriate inputs of AND gates. Often, the NOT gates are not specifically shown in the circuit. It is then assumed that the true and complemented values of variables are available. The logic circuit is usually shown as in Fig. 1.7(b). This type of circuit, designed using the SOP form of Boolean function as the starting point, is called a two-level AND–OR circuit because the first level consists of AND gates and the second level consists of OR gates.

### 1.5.2 OR–AND Circuits

An OR–AND circuit can be designed starting with the POS form for the function:

$$P = (X + Y + Z) \cdot (X + Y + Z') \cdot (X' + Y + Z) \cdot (X' + Y' + Z)$$

The design is carried out in three steps. The first two are shown in Fig. 1.8(a). The third step of including NOT gates is identical to that required in AND–OR circuit design.

In practice, the output functions are first simplified using the properties of Boolean algebra or the methods of Appendix B before drawing logic diagrams. We have ignored the simplification problem in the above example.

### 1.5.3 NAND–NAND and NOR–NOR Circuits

The NAND and NOR operations shown in Figure 1.3 are *universal* operations: each of the primitive operations AND, OR, and NOT can be realized using only NAND operators or only NOR operators. Figure 1.9 shows the realization of the three operations using only NAND gates. The theorems used in arriving at the simplified form of expressions are also identified in the figure. The NOR gate can be used in a similar way to realize all three primitive operations.

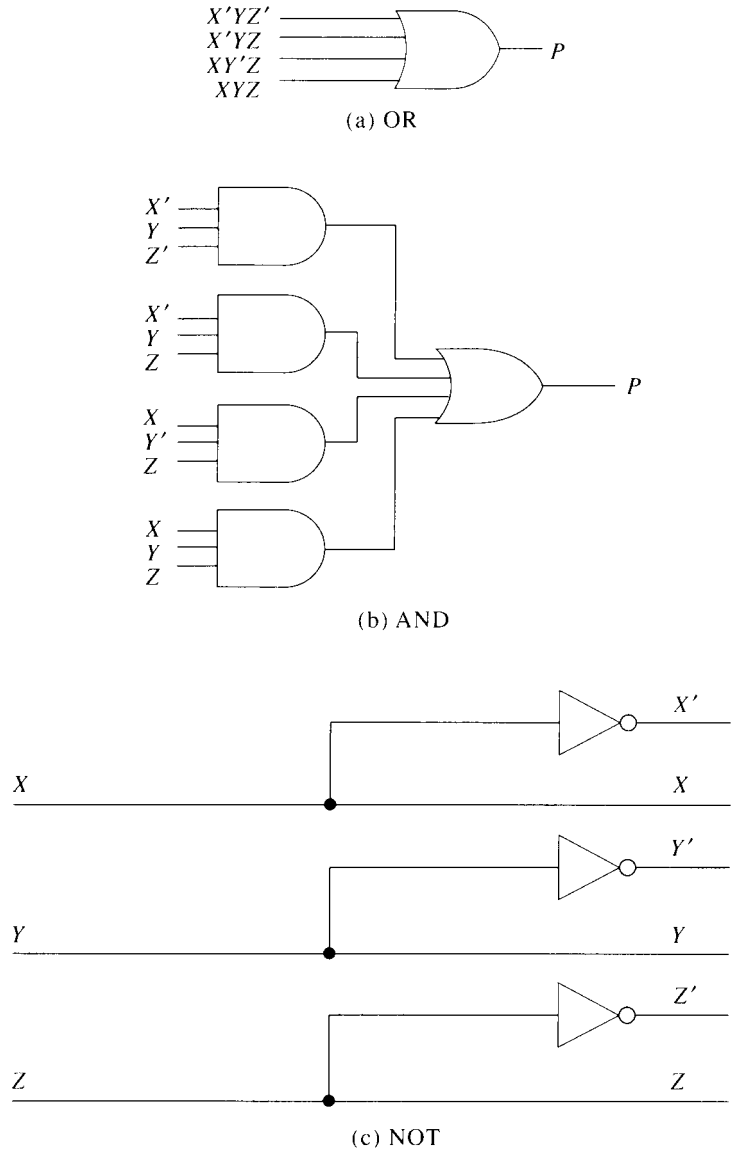



**Figure 1.7** AND–OR circuit

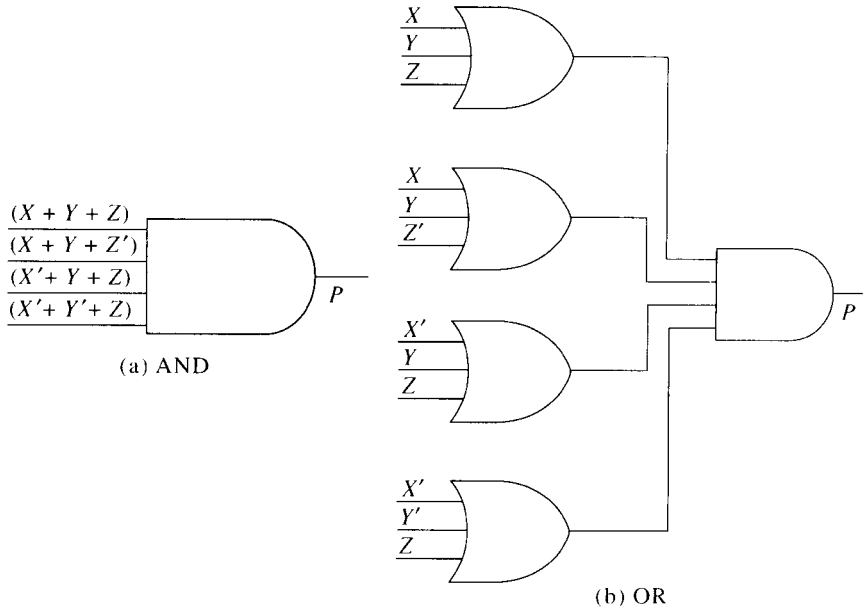



**Figure 1.8** OR–AND circuit

The universal character of NAND and NOR gates permits building of logic circuits using only one type of gate (i.e., NAND only or NOR only).

---

**Example 1.21** Figure 1.10 illustrates the transformation of an AND–OR circuit into a circuit consisting of only NAND gates. Each AND gate in the AND–OR circuit in (a) is replaced with two NAND gates (see Fig. 1.9). The circuit now has only NAND gates. There are some redundant gates in circuit (c). Gates 5 and 8 can be removed because these gates simply complement the input signal $(AB)'$ twice and hence are not needed. Similarly, gates 7 and 9 can be removed. The circuit in (d) is then a NAND–NAND circuit. The circuits in (a) and (d) are equivalent since both of them realize the same function $(AB + A'C)$.

---

A NAND–NAND implementation can thus be derived from an AND–OR circuit by simply replacing each gate in the AND–OR circuit with a NAND gate having the same number of inputs as that of the gate it replaces. A NOR–NOR implementation likewise can be obtained by start-

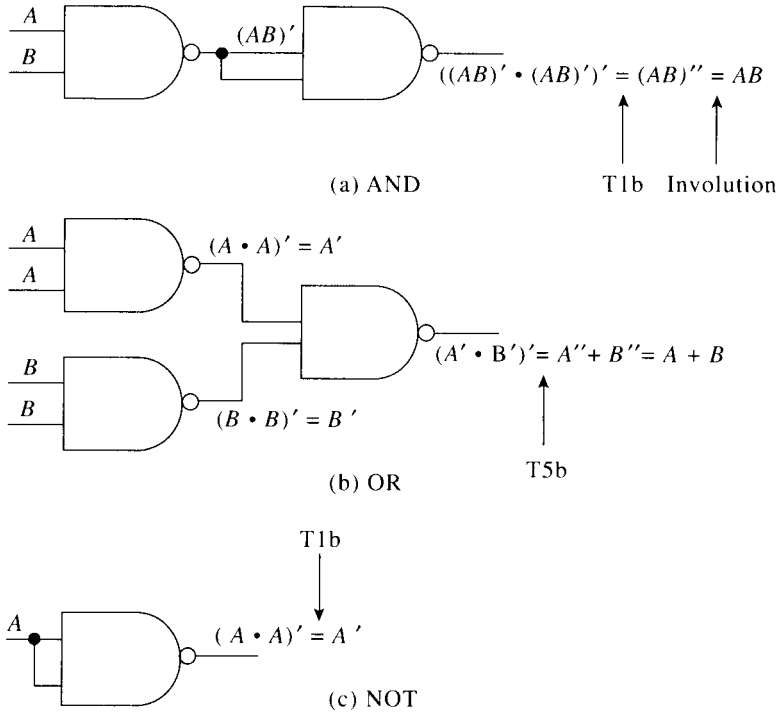
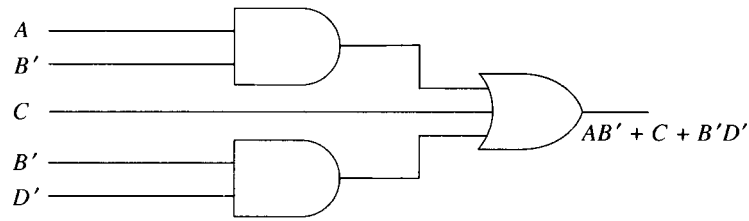
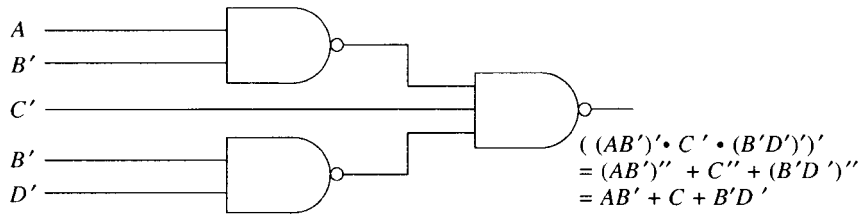



**Figure 1.9** Realization of primitive operations using NAND

ing with an OR–AND implementation and replacing each gate with a NOR gate. Any input literal feeding the second level directly must be inverted as shown by the following circuit:



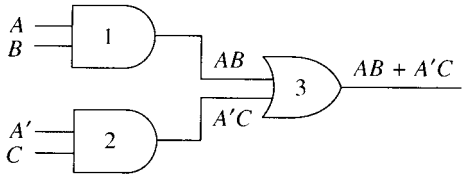

Here, $C$ is fed directly to the second level. Hence, it must be inverted to derive the correct NAND–NAND implementation, as shown in the circuit below:



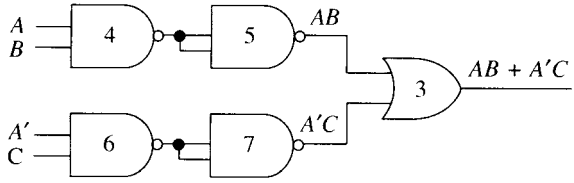

These implementations are feasible because the gates are available commercially in the form of *integrated circuits* (ICs) in packages containing several gates of the same type. Using the same types of gates eliminates the need for different types of ICs; and using the same type of ICs usually allows more efficient use of the ICs and reduces the IC package count. Further, the NAND and NOR circuits are primitive circuit configurations in major IC technologies, and the AND and OR gates are realized by complementing the outputs of NAND and NOR, respectively. Thus, NAND and NOR gates are less complex to fabricate and more cost efficient than the corresponding AND and OR gates.

We now summarize the combinational circuit design procedure:

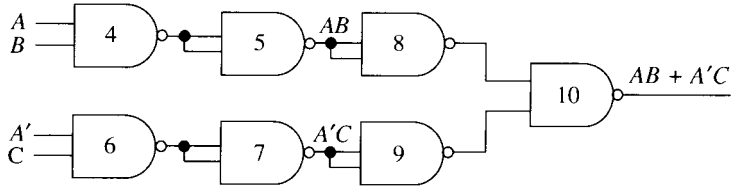
1. From the specification of the circuit function, derive the number of inputs and outputs required.
2. Derive the truth table.
3. If either an AND–OR or NAND–NAND form of circuit is required:
   a. Derive the SOP form of function for each output from the truth table.
   b. Simplify the output functions.
   c. Draw the logic diagram with the first level of AND (or NAND) gates and the second level of an OR (or NAND) gate, with appropriate number of inputs.
4. If either an OR–AND or NOR–NOR form of circuit is required:
   a. Derive the POS form of function for each output, from the truth table.
   b. Simplify the output functions if possible.
   c. Draw the logic diagram with the first level of OR (or NOR) gates and the second level of an AND (or NOR) gate, with appropriate number of inputs.



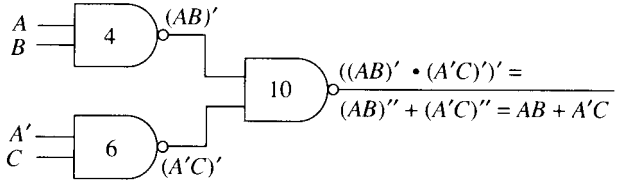

(a) AND–OR circuit




(b) Replace AND gates




(c) Replace OR gate




(d) Remove redundant gates (NAND–NAND circuit)

**Figure 1.10** NAND–NAND transformation

## 1.6 SOME POPULAR COMBINATIONAL CIRCUITS

The design of several most commonly used combinational logic circuits is given in this section. These are available as IC components. Details of some available components are given in the next section. We will illustrate these designs in AND–OR circuit forms; the other three forms can be derived from the truth tables given here.

### 1.6.1 Adders

Addition is the most common arithmetic operation performed by processors. If a processor has hardware capable of adding two numbers, the other three primitive arithmetic operations can also be performed using the addition hardware. Subtraction is performed by adding the subtrahend expressed in either 2s or 1s complement form to the minuend; multiplication is repeated addition of multiplicand to itself by multiplier number of times; and division is the repeated subtraction of divisor from dividend.

Consider the addition of two 4-bit numbers $A$ and $B$:

$$\begin{array}{rcccc} & c_2 & c_1 & c_0 & \\ \text{A:} & a_3 & a_2 & a_1 & a_0 \\ \text{B:} & b_3 & b_2 & b_1 & b_0 \\ \hline \text{SUM:} & c_3 & s_3 & s_2 & s_1 & s_0 \end{array}$$

Bits $a_0$ and $b_0$ are least significant bits (LSB); $a_3$ and $b_3$ are most significant bits (MSB). The addition is performed starting with the LSB position. Adding $a_0$ and $b_0$ will produce a SUM bit $s_0$ and a CARRY $c_0$. This CARRY $c_0$ is now used in the addition of the next significant bits $a_1$ and $b_1$, producing $s_1$ and $c_1$; this addition process is carried out through the MSB position.

A *half-adder* is a device that can add 2 bits producing a SUM bit and a CARRY bit as outputs. A *full adder* adds 3 bits, producing a SUM bit and a CARRY bit as its outputs. To add two $n$-bit numbers, we thus need one half-adder and $n - 1$ full adders. Figure 1.11 shows the half-adder and full adder arrangement to perform 4-bit addition. This is called a *ripple-carry adder* since the carry ripples through the stages of the adder starting at LSB to MSB. The time needed for this carry propagation is in proportion to the number of bits. Since the sum is of correct value only after the carry appears at MSB, the longer the carry propagation time, the slower the adder will be. There are several schemes to increase the speed of this adder. Some of them are discussed in Chapter 10.

Figure 1.12 shows the block diagram representations and truth tables for full adders and half-adders. From truth tables we can derive the SOP form functions for the outputs of the adders. They are:

Half-adder.

$$\begin{aligned} S_0 &= a_0' b_0 + a_0 b_0' \\ C &= a_0 b_0 \end{aligned} \tag{1.5}$$

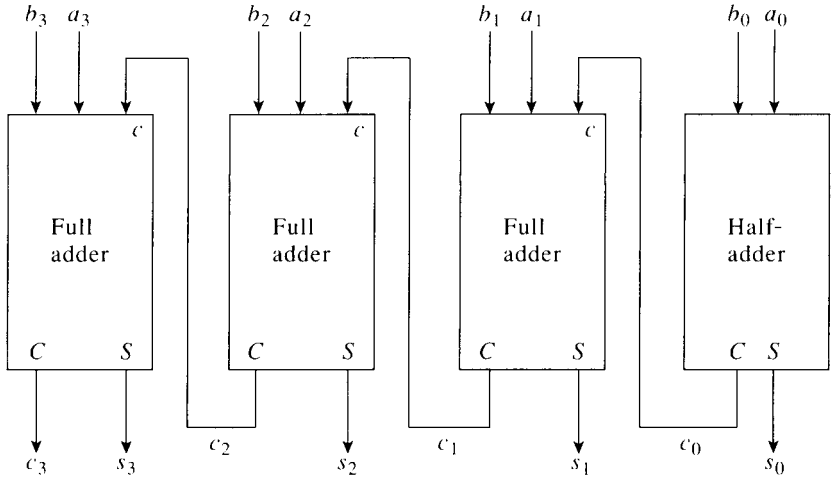
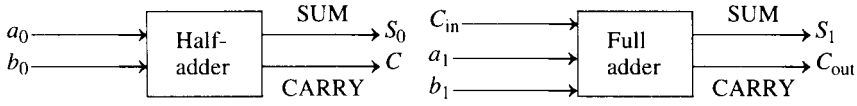



**Figure 1.11** A ripple-carry adder




Truth Table

| $a_0$ | $b_0$ | $S_0$ | $C$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

(a) Half-adder

Truth Table

| $C_{in}$ | $a_1$ | $b_1$ | $S_1$ | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

(b) Full adder

**Figure 1.12** Adders with truth tables

Figure 1.13 shows the circuit diagram.

Full adder.

$$\begin{aligned} S_1 &= C'_{\text{in}}a'_1b_1 + C'_{\text{in}}a_1b'_1 + C_{\text{in}}a'_1b'_1 + C_{\text{in}}a_1b_1 \\ C_{\text{out}} &= C'_{\text{in}}a_1b_1 + C_{\text{in}}a'_1b_1 + C_{\text{in}}a_1b'_1 + C_{\text{in}}a_1b_1 \end{aligned} \tag{1.6}$$

Figure 1.14 shows the circuit diagram.

The equation for the $C_{\text{out}}$ output of the full adder can be simplified using the theorems of Boolean algebra:

$$C_{\text{out}} = C'_{\text{in}}a_1b_1 + C_{\text{in}}a'_1b_1 + \underbrace{C_{\text{in}}a_1b'_1 + C_{\text{in}}a_1b_1}_{P4\text{b}}$$

$$= C'_{\text{in}}a_1b_1 + C_{\text{in}}a'_1b_1 + \underbrace{C_{\text{in}}a_1\underbrace{(b'_1 + b_1)}_{1}}_{} \qquad \text{P5a}$$

$$= C'_{\text{in}}a_1b_1 + \underbrace{C_{\text{in}}a'_1b_1 + C_{\text{in}}a_1}_{} \qquad \text{P1b}$$

$$= C'_{\text{in}}a_1b_1 + C_{\text{in}}\underbrace{(a'_1b_1 + a_1)}_{\text{T4a}} \qquad \text{P4b}$$

$$= C'_{\text{in}}a_1b_1 + C_{\text{in}}(b_1 + a_1)$$

$$= \underbrace{C'_{\text{in}}a_1b_1 + C_{\text{in}} \cdot b_1}_{} + C_{\text{in}} \cdot a_1 \qquad \text{P4b}$$

$$= \underbrace{(C'_{\text{in}}a_1 + C_{\text{in}})}_{\text{T4a}}b_1 + C_{\text{in}}a_1 \qquad \text{P4b}$$

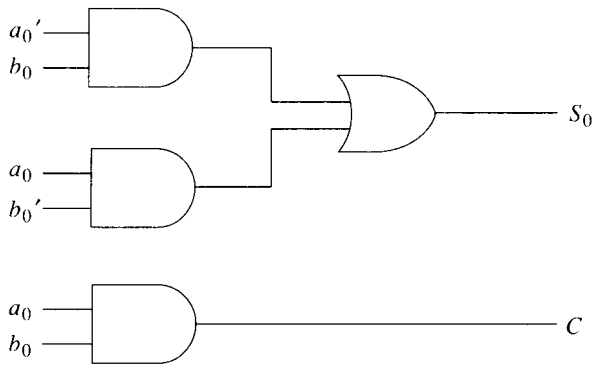



**Figure 1.13** Half-adder circuits

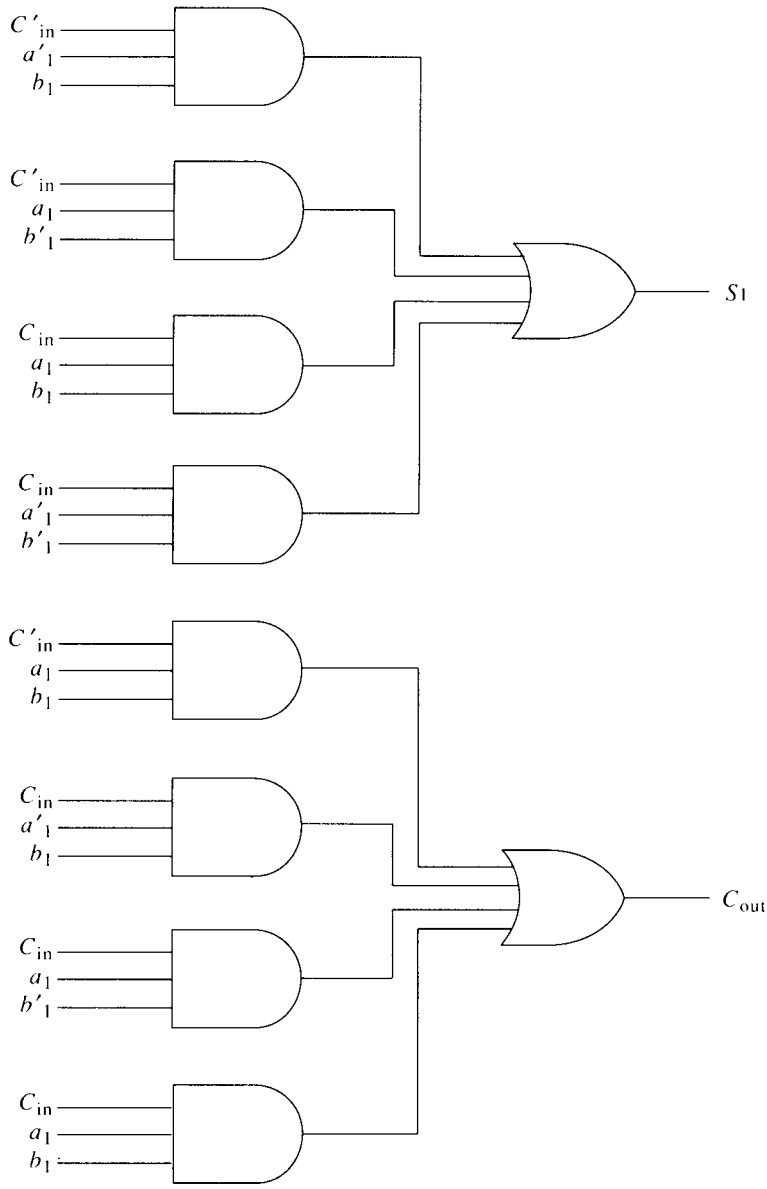


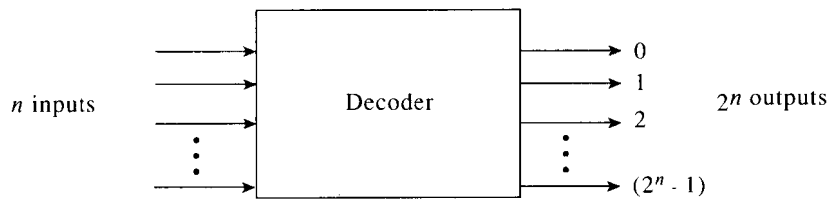

**Figure 1.14** Full adder circuits

$$= (a_1 + C_{in})b_1 + C_{in}a_1 \qquad \text{P4b}$$
$$= a_1b_1 + C_{in}b_1 + C_{in}a_1$$

This equation has only 6 literals compared to the 12 literals of the original equation. This simplified equation can be realized with three 2-input AND gates and one 3-input OR gate. Such simplifications are usually performed while building a circuit using gates. Appendix B gives two more simplification procedures that are more mechanical to perform than the algebraic procedure shown above.

### 1.6.2 Decoders

A *code word* is a string of a certain number of bits. Appendix A lists the most common codes. The hexadecimal system, for example, uses a 4-bit code word for each digit 0, 1, 2, 3, $\cdots$, D, E, F. An $n$-bit binary string can take $2^n$ combinations of values. An $n$-to-$2^n$ decoder is a circuit that converts the $n$-bit input data into $2^n$ outputs (at the maximum). At any time only one output line corresponding to the combination on the input lines will be 1; all the other outputs will be 0. The outputs are usually numbered from 0 to $(2^n - 1)$. If, for example, the combination on the input of a 4-to-$2^4$ decoder is 1001, only output 9 will be 1; all other outputs will be 0.




It is not usually necessary to draw a truth table for a decoder. There would be a single 1 in each output column of the truth table and the product (or SUM) term corresponding to that 1 could be easily derived. Figure 1.15 shows the circuit diagram of a 3-to-8 decoder. The 3 inputs are designated A, B, and C, with C as the least significant bit (LSB). The outputs are numbered 0 through 7.

### 1.6.3 Code Converters

A code converter translates an input code word into an output bit pattern corresponding to a new code word. A decoder is a code converter that changes an $n$-bit code word into a $2^n$-bit code word. We will illustrate the

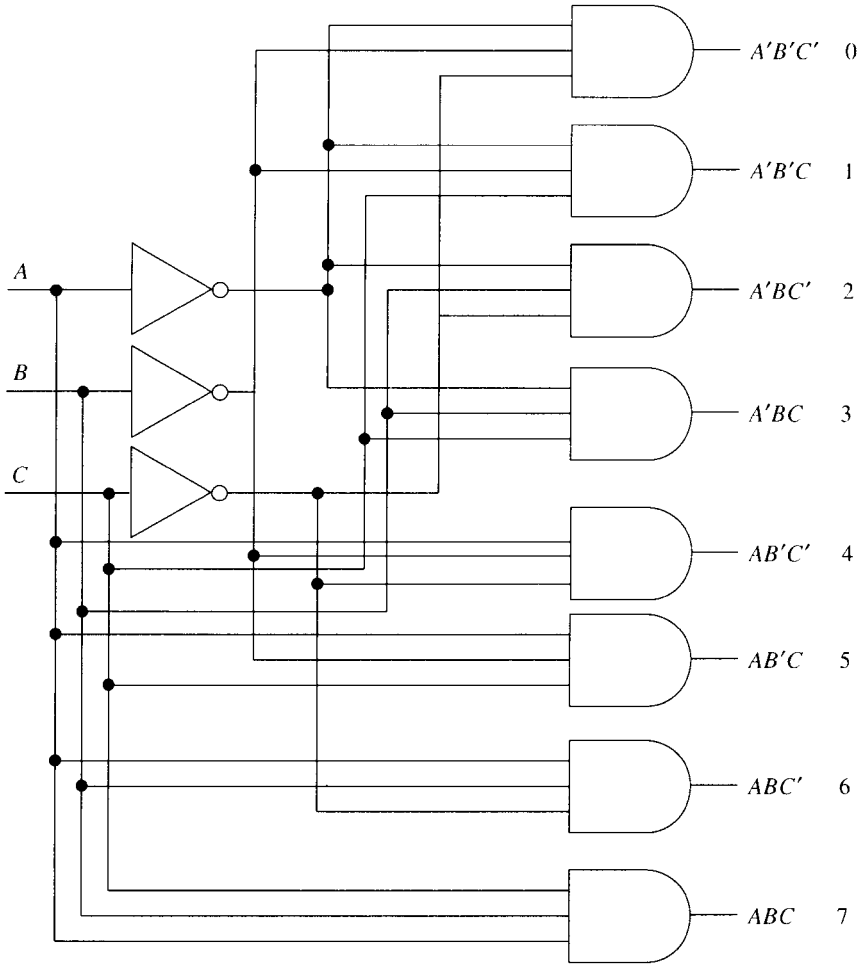



**Figure 1.15** A 3-to-8 decoder circuit

design of a circuit that converts the Binary Coded Decimal (BCD) into Excess-3 code. The truth table is shown in Fig. 1.16. Both BCD and Excess-3 are 4-bit codes. BCD code extends from 0 to 9 and there are 16 combinations of 4 bits, so the last 6 combinations are not used in BCD. They will never occur as inputs to the circuit. Hence, we DON'T CARE what happens to the outputs, for these input values. These DON'T CARE conditions are shown as "*d*" on the truth table. They can be used to advantage in simplifying functions because they can be either a 0 or a 1 for our convenience. From Fig. 1.16:

| | BCD Inputs | | | | Excess-3 Output | | | |
|---|---|---|---|---|---|---|---|---|
| Decimal | $A$ | $B$ | $C$ | $D$ | $W$ | $X$ | $Y$ | $Z$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Not used | 1 | 0 | 1 | 0 | $d$ | $d$ | $d$ | $d$ |
| | 1 | 0 | 1 | 1 | $d$ | $d$ | $d$ | $d$ |
| | 1 | 1 | 0 | 0 | $d$ | $d$ | $d$ | $d$ |
| | 1 | 1 | 0 | 1 | $d$ | $d$ | $d$ | $d$ |
| | 1 | 1 | 1 | 0 | $d$ | $d$ | $d$ | $d$ |
| | 1 | 1 | 1 | 1 | $d$ | $d$ | $d$ | $d$ |

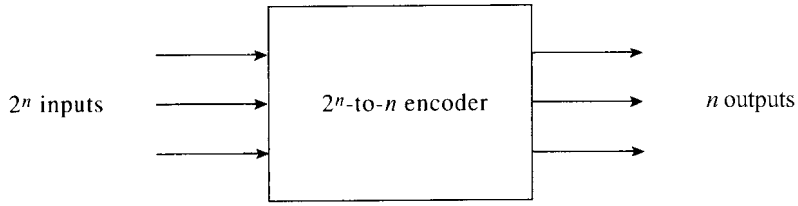
**Figure 1.16** Truth table for BCD to Excess-3 decoder

$$W = A'BC'D + A'BCD' + A'BCD + AB'C'D' + AB'C'D$$
$$X = A'B'C'D + A'B'CD' + A'B'CD + A'BC'D' + AB'C'D$$
$$Y = A'B'C'D' + A'B'CD + A'BC'D' + A'BCD + AB'C'D'$$
$$Z = A'B'C'D' + A'B'CD' + ABC'D' + A'BCD' + AB'C'D'$$

In multiple-output designs such as this, it is possible that two or more output functions contain the same product term. For example, $A'B'C'D'$ appears in both $Y$ and $Z$. It is not necessary to implement this product term twice. The output of the gate realizing this product term can be fanned out to be used in the realization of both $Y$ and $Z$. Further simplification of these functions is possible when the DON'T CARE conditions are taken into account. Appendix B shows such a minimization for the BCD to Excess-3 code converter.

### 1.6.4 Encoders

An encoder generates an $n$-bit code word as a function of the combination of values on its input. At the maximum, there can be $2^n$ inputs.




The design of an encoder is executed by first drawing a truth table that shows the $n$-bit output needed for each of the $2^n$ combinations of inputs. The circuit diagrams are then derived for each output bit.

---

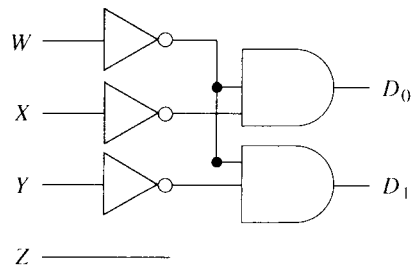
**Example 1.22** A partial truth table for a 4-to-2 line encoder is shown in Fig. 1.17(a). Although there are 16 combinations of 4 inputs, only 4 are used because the 2-bit output supports only 4 combinations. The output combinations identify which of the four input lines is at 1 at a particular time. The output functions as can be seen from this truth table are

$$\begin{aligned} D_0 &= W'X', \text{ and} \\ D_1 &= W'Y' \end{aligned} \tag{1.7}$$

These functions may be simplified by observing that it is sufficient to have $W = 0$ and $X = 0$ for $D_0$ to be 1, no matter what the values of $Y$ and $Z$ are. Similarly, $W = 0$ and $Y = 0$ are sufficient for $D_1$ to be 1. Such observations, although not always straightforward, help in simplifying functions, thus reducing the amount of hardware needed. Alternatively, the truth table in Fig. 1.17(a) can be completed by including the remaining 12 input combinations and entering don't cares for the outputs corresponding to those

| Inputs | | | | Outputs | |
|---|---|---|---|---|---|
| $W$ | $X$ | $Y$ | $Z$ | $D_0$ | $D_1$ |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |

(a) Truth table



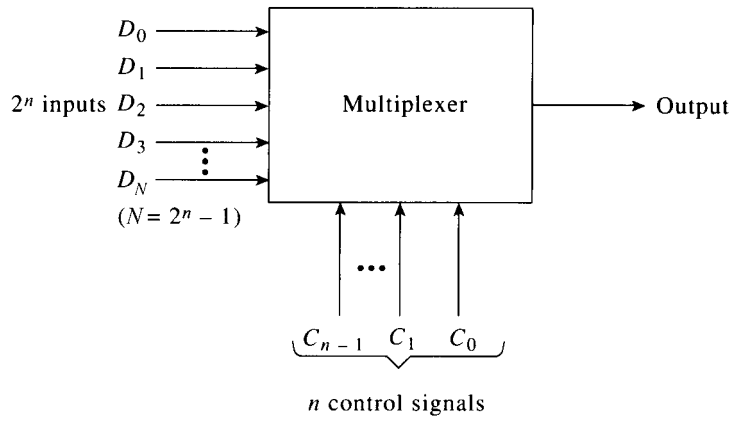

(b) Circuit

**Figure 1.17** A 4-to-2 encoder

inputs. $D_0$ and $D_1$ can then be derived from the truth table and simplified. Figure 1.17(b) shows the circuit diagram for the 4-to-2 encoder.

### 1.6.5 Multiplexers

A multiplexer is a switch that connects one of its several inputs to the output. A set of $n$ control inputs is needed to select one of the $2^n$ inputs that is to be connected to the output.




**Example 1.23** The operation of a multiplexer with 4 inputs ($D_0 - D_3$) and hence 2 control signals ($C_0$, $C_1$) can be described by the following table:

| $C_1$ | $C_0$ | Output |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

Although there are 6 inputs, a complete truth table with $2^6$ rows is not required for designing the circuit, since the output simply assumes the value of one of the 4 inputs depending on the control signals $C_1$ and $C_0$. That is,

$$\text{Output} = D_0 \cdot C_1' C_0' + D_1 \cdot C_1' C_0 + D_2 \cdot C_1 C_0' + D_3 \cdot C_1 C_0 \tag{1.8}$$

The circuit for realizing the above multiplexer is shown in Fig. 1.18.

---

Each of the inputs $D_0$, $D_1$, $D_2$, and $D_3$ and the output in this multiplexer circuit are single lines. If the application requires that the data lines to be multiplexed have more than one bit each, the above circuit has to be duplicated once for each bit of data.

### 1.6.6 Demultiplexers

A demultiplexer has one input and several outputs. It switches (connects) the input to one of its outputs based on the combination of values on a set of control (select) inputs. If there are $n$ control signals, there can be a maximum of $2^n$ outputs.

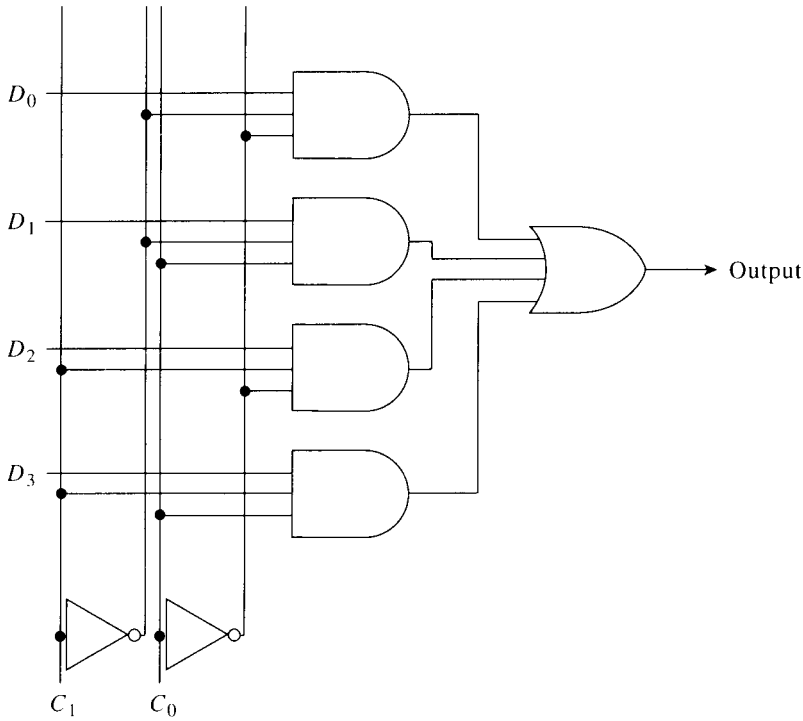
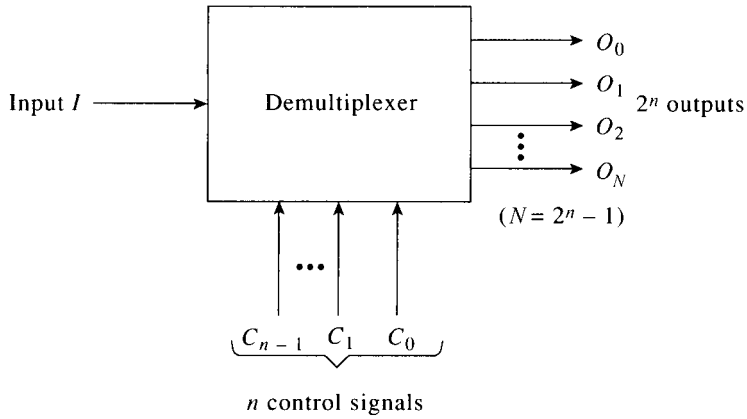



**Figure 1.18** A 4-to-1 multiplexer




**Example 1.24** The operation of a demultiplexer with four outputs ($O_0 - O_3$) and hence two control signals ($C_1, C_0$) can be described by the following table:

| $C_1$ | $C_0$ | $O_3$ | $O_2$ | $O_1$ | $O_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

It is not necessary to draw a truth table with eight rows for this circuit, although there are three inputs to it, since the input $I$ is directed to only one of the four outputs based on the four combinations of values on the control signals $C_1$ and $C_0$. Thus,

$$O_0 = IC_1'C_0' \quad O_1 = IC_1'C_0$$
$$O_2 = IC_1C_0' \quad O_3 = IC_1C_0$$

A typical application for a multiplexer is to connect one of the several input devices (as selected by a device number) to the input of a computer system. A demultiplexer can be used to switch the output of the computer on to one of the several output devices.

## 1.7 INTEGRATED CIRCUITS

So far in this chapter we have concentrated only on the functional aspects of gates and logic circuits in terms of manipulating binary signals. A gate is an electronic circuit made up of transistors, diodes, resistors, capacitors, and other components interconnected to realize a particular function. In this section, we will expand our understanding of gates and circuits to the electronic level of detail.

At the current state of digital hardware technology, the logic designer combines ICs that perform specific functions to realize functional logic units. An IC is a small slice of silicon semiconductor crystal called a *chip*, on which the discrete electronic components mentioned above are chemically fabricated and interconnected to form gates and other circuits. These circuits are accessible only through the pins attached to the chip. There will be one pin for each input signal and one for each output signal of the circuit fabricated on the IC. The chip is mounted in either a metallic or a plastic package. Various types of packages, such as Dual-In-Line Package (DIP) and flat package, are used. DIP is the most widely used package. The number of pins varies from 8 to 64. Each IC is given a numeric designation (printed on the package), and the IC manufacturer's catalog provides the functional and elctronic details on the IC.

Each IC contains one or more gates of the same type. The logic designer combines ICs that perform specific functions to realize functional logic units. As such, the electronic-level details of gates usually are not needed to build efficient logic circuits. But as logic circuit complexity increases, the electronic characteristics become important in solving the timing and loading problems in the circuit.

Figure 1.19 shows the details of an IC that comes from the popular TTL (transistor-transistor logic) family of ICs. It has the numeric designation of 7400 and contains four two-input NAND gates. There are 14 pins. Pin 7 (ground) and pin 14 (supply voltage) are used to power the IC. Three

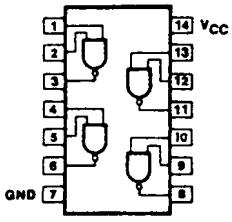



**Figure 1.19** A typical IC (TTL 7400)

pins are used by each gate (two for the input and one for the output). On all ICs, a "notch" on the package is used to reference pin numbers. Pins are numbered counterclockwise starting from the notch.

*Digital* and *linear* are two common classifications of ICs. Digital ICs operate with binary signals, while linear ICs operate with continuous signals. In this book, we will be dealing only with digital ICs.

Because of the advances in IC technology, it is now possible to fabricate a large number of gates on a single chip. According to the number of gates it contains, an IC can be classified as a small-, medium-, large- or very-large-scale circuit. An IC containing a few gates (approximately 10) is called a small-scale integrated (SSI) circuit. A medium-scale integrated (MSI) circuit has a complexity of around 100 gates and typically implements an entire function, such as an adder or a decoder, on a chip. An IC with a complexity of more than 100 gates is a large-scale integrated (LSI) circuit, while a very-large-scale integrated (VLSI) circuit contains thousands of gates.

There are two broad categories of IC technology, one based on *bipolar* transistors (i.e., p-n-p or n-p-n junctions of semiconductors) and the other based on the *unipolar* metal oxide semiconductor field effect transistor (MOSFET). Within each technology, several logic families of ICs are available. The popular bipolar logic families are TTL and emitter-coupled logic (ECL). P-channel MOS (PMOS), N-channel MOS (NMOS), and complementary MOS (CMOS) are all popular MOS logic families. A new family of ICs based on gallium arsenide has been introduced recently. This technology has the potential of providing ICs that are faster than ICs in silicon technology.

In the following discussion, we will examine functional-level details of ICs. These details are adequate to build circuits using ICs. Various performance characteristics of ICs will be introduced, without electronic-level justification.

In addition to the details of the type provided in Figure 1.19, the IC manufacturer's catalog contains such information as voltage ranges for each logic level, fan-out, propagation delays, and so forth, for each IC. In this section, we will introduce the major symbols and notation used in the IC catalogs and describe the most common characteristics in selecting and using ICs.

### 1.7.1 Positive and Negative Logic

As mentioned earlier, two distinct voltage levels are used to designate logic values 1 and 0, and in practice these voltages levels are ranges of voltages, rather than fixed values. Figure 1.20 shows the voltage levels used in the

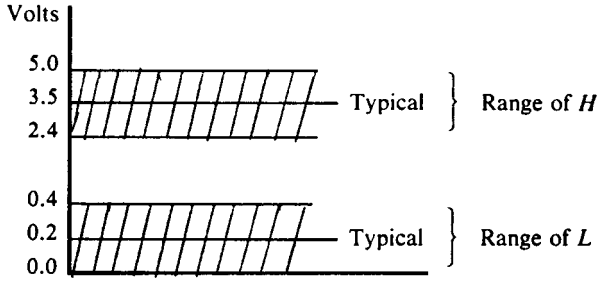



**Figure 1.20** TTL voltage levels

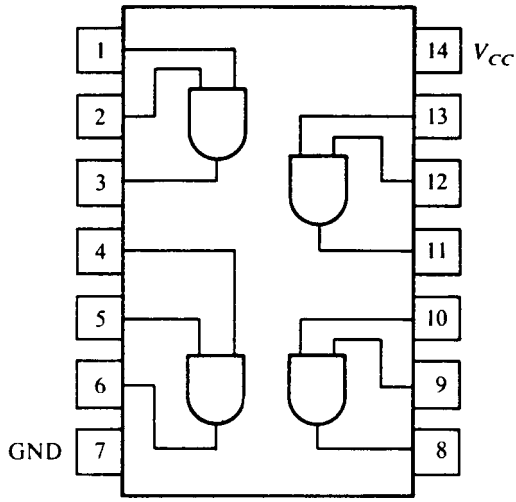
TTL technology: the high level corresponds to the range of 2.4 to 5 V and the low level corresponds to 0 to 0.4 V. These two levels are designated *H* and *L*.

In general, once the voltage levels are selected, the assignment of 1 and 0 to those levels can be arbitrary. In the so-called *positive logic* system, the higher of the two voltages denotes logic-1, and the lower value denotes logic-0. In the *negative logic* system, the designations are the opposite. The following table shows the two possible logic value assignments.

| | Positive logic | Negative logic |
|---|---|---|
| Logic-1 | *H* | *L* |
| Logic-0 | *L* | *H* |

Note that *H* and *L* can both be positive, as in TTL, or both negative, as in ECL ($H = -0.7$ to $-0.95$ V, $L = -1.9$ to $-1.5$ V). It is the assignment of the relative magnitudes of voltages to logic-1 and logic-0 that determines the type of logic, rather than the polarity of the voltages.

Because of these "dual" assignments, a logic gate that implements an operation in the positive logic system implements its dual operation in the negative logic system. IC manufacturers describe the function of gates in terms of *H* and *L*, rather than logic-1 and logic-0. As an example, consider the TTL 7408 IC, which contains four two-input AND gates (Fig. 1.21(a)). The function of this IC as described by the manufacturer in terms of *H* and *L* is shown in the voltage table. Using positive logic, the voltage table (b) can be converted into the truth table (c) representing the positive logic AND, as shown by the gate (d). Assuming negative logic, table (b) can be converted into truth table (e), which is the truth table for OR. The negative logic OR



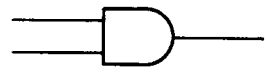

(a) TTL 7408

| X | Y | Z |
|---|---|---|
| L | L | L |
| L | H | L |
| H | L | L |
| H | H | H |

(b) Voltage table

| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(c) Positive logic truth table



(d) Positive logic AND

| X | Y | Z |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

(e) Negative logic truth table

(f) Negative logic OR

**Figure 1.21** Positive and negative logic

gate is shown in (f). The half-arrows on the inputs and the output designate them to be negative logic values. Note that the gates in (d) and (f) both correspond to the same physical gate but function either as positive logic AND or as negative logic OR. Similarly, it can be shown that the following dual operations are valid:

| Positive logic | Negative logic |
|---|---|
| OR | AND |
| NAND | NOR |
| NOR | NAND |
| EXCLUSIVE-OR | EQUIVALENCE |
| EQUIVALENCE | EXCLUSIVE-OR |

We will assume the positive logic system throughout this book and as such will not use negative logic symbolism. In practice, a designer may combine the two logic notations in the same circuit (mixed logic), so long as the signal polarities are interpreted consistently.

### 1.7.2 Signal Inversion

We have used a "bubble" in NOT, NOR, and NAND gate symbols to denote signal inversion (complementation). This bubble notation can be extended to any logic diagram. Some examples are shown in Fig. 1.22. The NOT gate symbol in (a) implies that when the input $X$ is asserted (e.g., at $H$), the output is low ($L$). The input is said to be *active-high* and the output is said to be *active-low*. An alternative symbol for a NOT gate, with an active-low input, is shown in (b). An AND gate with active-low inputs is shown in (c). The output of this gate is high only when both inputs are low. Note that this is the INVERT–AND or a NOR gate. A typical IC with four inputs and one output is shown in (d). Input $A$ and output $E$ are active-low, and inputs $B$, $C$, and $D$ are active-high. An $L$ on input $A$ would appear as an $H$ internal to the IC, and an $H$ corresponding to $E$ internal to

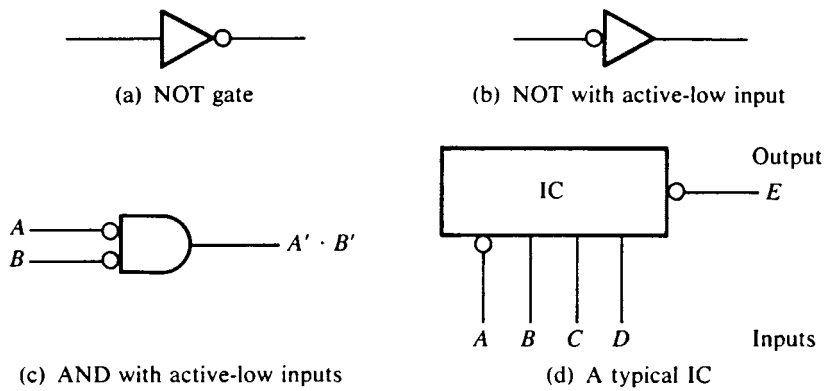



**Figure 1.22** Bubble notation

the IC would appear as an *L* external to the IC. That is, an active-low signal is active when it carries a low logic value, while an active-high signal is active when it carries a high logic value. A bubble in the logic diagram indicates an active-low input or output. For example, the BCD-to-decimal decoder IC (7442) shown in Fig. 1.23 has four active-high inputs corresponding to the four input BCD bits and ten active-low outputs. Only the output corresponding to the decimal value of the input BCD number is active at any time. That is, the output corresponding to the decimal value of the BCD input to the IC will be *L* and all other outputs will be *H*.

When two or more ICs are used to the circuit, the active-low and active-high designations of input and output signals must be observed for the proper operation of the circuit, although ICs of the same logic family generally have compatible signal-active designations.

It is important to distinguish between the negative logic designation (half-arrow) and the active-low designation (bubble) in a logic diagram. These designations are similar in effect, as shown by the gate symbols in Fig. 1.24, and as such can be replaced by each other. In fact, as shown in (c) and (d) on the figure a half-arrow following the bubble cancels the effect of the bubble on the signal, and hence both the half-arrow and the bubble can be removed. Then, the inputs and outputs of the gate are of different polarity. For example, if the half-arrow and the bubble are removed from the output of the negative logic NOR gate in (c), the inputs represent negative logic and the outputs represent positive logic polarities.

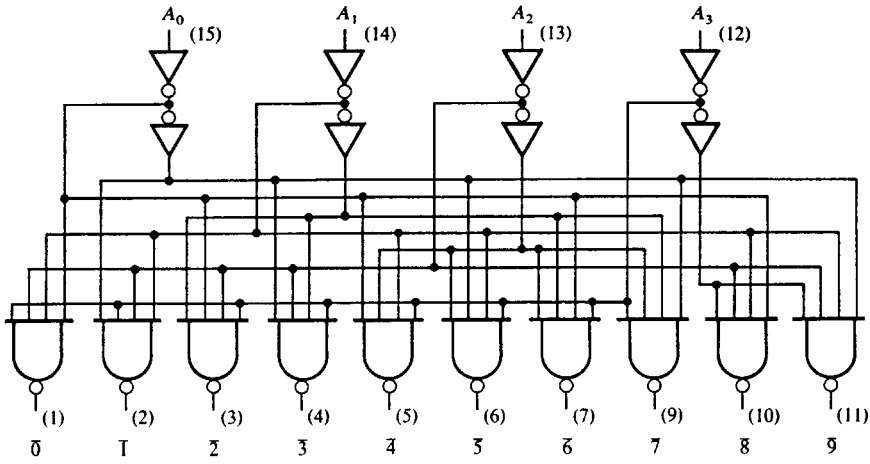
### 1.7.3 Other Characteristics

Other important characteristics to be noted while designing with ICs are the active-low and active-high designations of signals, voltage polarities, and low and high voltage values, especially when ICs of different logic families are used in the circuit. ICs of the same family are usually compatible with respect to these characteristics. Special ICs to interface circuits built out of different IC technologies are also available.

Designers usually select a logic family on the basis of the following characteristics:

1. Speed
2. Power dissipation
3. Fan-out
4. Availability
5. Noise immunity (noise margin)
6. Temperature range
7. Cost.

Logic diagram


$A_0$ (15)
$A_1$ (14)
$A_2$ (13)
$A_3$ (12)
(1) (2) (3) (4) (5) (6) (7) (9) (10) (11)
$\overline{0}$ $\overline{1}$ $\overline{2}$ $\overline{3}$ $\overline{4}$ $\overline{5}$ $\overline{6}$ $\overline{7}$ $\overline{8}$ $\overline{9}$

( ) = Pin number
$V_{CC}$ = Pin 16
GND = Pin 8

LD02750S

**Function table**

| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $\overline{0}$ | $\overline{1}$ | $\overline{2}$ | $\overline{3}$ | $\overline{4}$ | $\overline{5}$ | $\overline{6}$ | $\overline{7}$ | $\overline{8}$ | $\overline{9}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | L | L | L | L | H | H | H | H | H | H | H | H | H |
| L | L | L | H | H | L | H | H | H | H | H | H | H | H |
| L | L | H | L | H | H | L | H | H | H | H | H | H | H |
| L | L | H | H | H | H | H | L | H | H | H | H | H | H |
| L | H | L | L | H | H | H | H | L | H | H | H | H | H |
| L | H | L | H | H | H | H | H | H | L | H | H | H | H |
| L | H | H | L | H | H | H | H | H | H | L | H | H | H |
| L | H | H | H | H | H | H | H | H | H | H | L | H | H |
| H | L | L | L | H | H | H | H | H | H | H | H | L | H |
| H | L | L | H | H | H | H | H | H | H | H | H | H | L |
| H | L | H | L | H | H | H | H | H | H | H | H | H | H |
| H | L | H | H | H | H | H | H | H | H | H | H | H | H |
| H | H | L | L | H | H | H | H | H | H | H | H | H | H |
| H | H | L | H | H | H | H | H | H | H | H | H | H | H |
| H | H | H | L | H | H | H | H | H | H | H | H | H | H |
| H | H | H | H | H | H | H | H | H | H | H | H | H | H |

H = HIGH voltage levels
L = LOW voltage levels

**Figure 1.23** TTL 7442 BCD-to-decimal decoder

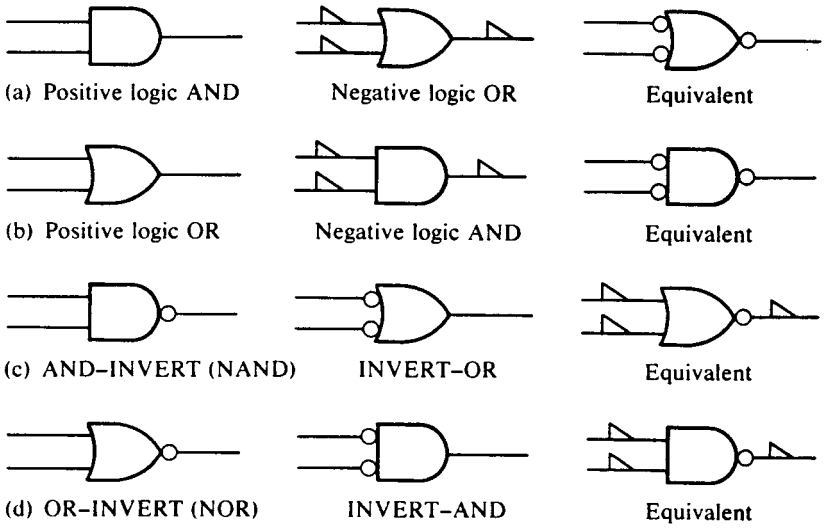



**Figure 1.24** Equivalent symbols

*Power dissipation* is proportional to the current that is drawn from the power supply. The current is inversely proportional to the equivalent resistance of the circuit, which depends on the values of individual resistors in the circuit, the load resistance, and the operating point of the transistors in the circuit. To reduce the power dissipation, the resistance should be increased. However, increasing the resistance increases the rise times of the output signal. The longer the rise time, the longer it takes for the circuit output to "'switch" from one level to the other. That is, the circuit is slower. Thus, a compromise between the speed (i.e., switching time) and power dissipation is necessary. The availability of various versions of TTL, for instance, is the result of such compromises.

A measure used to evaluate the performance of ICs is the *speed–power product*. The smaller this product, the better the performance. The speed–power product of a standard TTL with a power dissipation of 10 mW and a propagation delay of 10 ns is 100 pJ.

The *noise margin* of a logic family is the deviation in the $H$ and $L$ ranges of the signal that is tolerated by the gates in the logic family. The circuit function is not affected so long as the noise margins are obeyed. Figure 1.25 shows the noise margins of TTL. The output voltage level of the gate stays either below $V_{L\max}$ or above $V_{H\min}$. When this output is connected to the input of another gate, the input treats any voltage

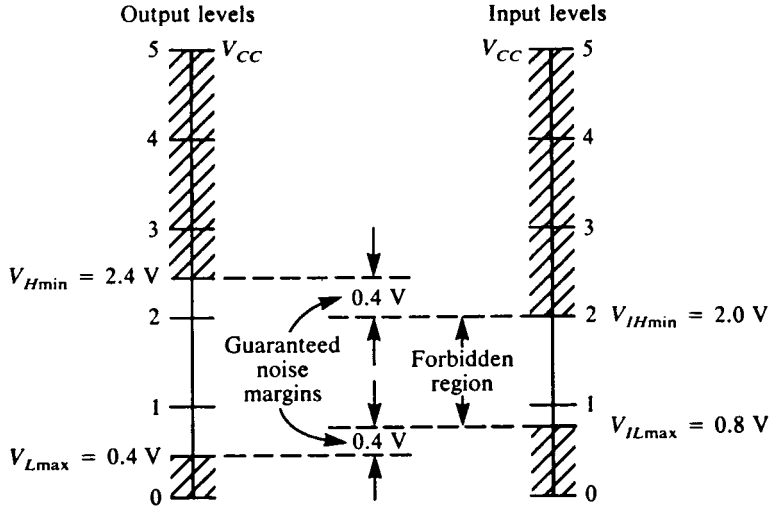



**Figure 1.25** TTL noise margins

above $V_{IH\min}$ (2.0 V) as high and any voltage below $V_{IL\min}$ (0.8 V) as low. Thus, TTL provides a *guaranteed noise margin* of 0.4 V. A supply voltage $V_{CC}$ of 5 V is required. Depending on the IC technology, as the load on the gate is increased (i.e., as the number of inputs connected to the output is increased), the output voltage may enter the *forbidden region*, thereby contributing to the improper operation. Care must be taken to ensure that the output levels are maintained by obeying the fan-out constraint of the gate or by using gates with special outputs or with higher fan-out capabilities.

The *fan-out* of a gate is the maximum number of inputs of other gates that can be connected to its output without degrading the operation of the gate. The fan-out is a function of the current sourcing and sinking capability of the gate. When a gate provides the driving current to gate inputs connected to its output, the gate is a current *source*, while it is a current *sink* when the current flows from the gates connected to it into its output. It is customary to assume that the driving and driven gates are of the same IC technology. Fan-out of gates in one technology with respect to gates of another technology can be determined according to the current sourcing and sinking capabilities of the gates in both technologies.

When the output of a gate is high, it acts as a current source for the inputs it is driving (see Fig. 1.26). As the current increases, the output

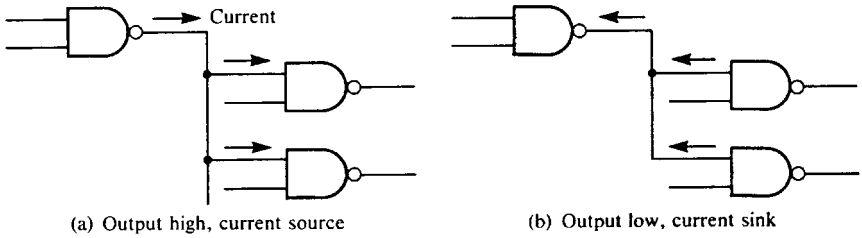



**Figure 1.26** Fan-out

voltage decreases and may enter the forbidden region. The driving capability is thus limited by the voltage drop. When the output is low, the driving gate acts as a current sink for the inputs. The maximum current that the output transistor can sink is limited by the *heat dissipation* limit of the transistor. Thus, the fan-out is the minimum of these two driving capabilities.

A standard TTL gate has a fan-out of 10. A standard TTL buffer can drive up to 30 standard TTL gates. Various versions of TTL are available. Depending on the version, the fan-out ranges between 10 and 20. Typical fan-out of ECL gates is 25, and that of CMOS is 50.

The popularity of the IC family helps the availability. The cost of ICs comes down when they are produced in large quantities. Therefore, very popular ICs are generally available. The other availability measure is the number of types of ICs in the family. The availability of a large number of ICs in the family makes for more design flexibility.

Temperature range of operation is an important consideration, especially in such environments as military applications, automobiles, and so forth, where temperature variations are severe. Commercial ICs operate in the temperature range of $0^\circ$ to $70^\circ$C, while ICs for military applications can operate in the temperature range of $-55^\circ$ to $+125^\circ$C.

The cost of an IC depends on the quantity produced. Popular off-the-shelf ICs have become very inexpensive. As long as off-the-shelf ICs are used in a circuit, the cost of the circuit's other components (e.g., the circuit board, connectors, and interconnections) currently is higher than that of the ICs themselves.

Circuits that are required in very large quantities can be *custom-designed* and fabricated as ICs. Small quantities do not justify the cost of custom design and fabrication. There are several types of *programmable* ICs that allow a *semicustom* design of ICs for special applications.

Table 1.1 summarizes the characteristics of the popular IC technologies.

**Table 1.1** Characteristics of Some Popular Logic Families

| Characteristic | TTL | ECL | CMOS |
|---|---|---|---|
| Supply voltage (V) | 5 | −5.2 | 3 to 18 |
| High-level voltage (V) | 2 to 5 | −0.95 to −0.7 | 3 to 18 |
| Low-level voltage (V) | 0 to 0.4 | −1.9 to −1.5 | 0 to 0.5 |
| Propagation delay (ns) | 5 to 10 | 1 to 2 | 25 |
| Fan-out | 10 to 20 | 25 | 50 |
| Power dissipation (mW) per gate | 2 to 10 | 25 | 0.1 |

### 1.7.4 Special Outputs

Several ICs with special characteristics are available and useful in building logic circuits. We will briefly examine these special ICs in this section, at a functional level of detail.

A gate in the logic circuit is said to be "loaded" when it is required to drive more inputs than its fan-out. Either the loaded gate is replaced with a gate of the same functionality but of a higher fan-out (if available in the IC family) or a *buffer* is connected to its output. The ICs designated as buffers (or "drivers") provide a higher fan-out than a regular IC in the logic family. For example, the following are some of the TTL 7400 series of ICs that are designated as buffers:

7406 Hex inverter buffer/driver.
7407 Hex buffer/driver (noninverting).
7433 Quad two-input NOR buffer.
7437 Quad two-input NAND buffer.

These buffers can drive approximately 30 standard TTL loads, compared to a fan-out of 10 for nonbuffer ICs.

In general, the outputs of two gates cannot be connected without damaging those gates. Gates with two special types of outputs are available that, under certain conditions, allow their outputs to be connected to realize an AND or an OR function of the output signals. The use of such gates thus results in reduced complexity of the circuit. The need for such gates is illustrated by Example 1.25.

---

**Example 1.25** We need to design a circuit that connects one of the four inputs $A$, $B$, $C$, and $D$ to the output $Z$. There are four control inputs ($C_1$, $C_2$, $C_3$, and $C_4$) that determine whether $A$, $B$, $C$, or $D$ is connected to $Z$, respectively. It is also known that only one of the inputs is connected to

the output at any given time. That is, only one of the control inputs will be active at any time. The function of this circuit can thus be represented as

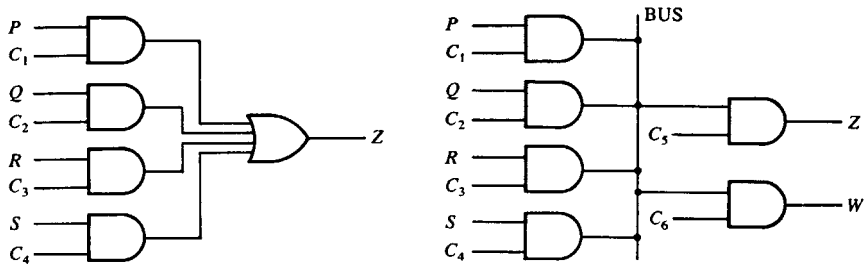
$$Z = P \cdot C_1 + Q \cdot C_2 + R \cdot C_3 + S \cdot C_4$$

Figure 1.27(a) shows the AND–OR circuit implementation of this function.

---

If each of the AND gates in (a) are such that their outputs can be connected to form an OR function, the four-input OR gate can be eliminated from the circuit. Furthermore, as the number of inputs increases (along with the corresponding increase in control inputs), the circuit can be expanded, simply by connecting the additional AND gate outputs to the common output connection. This way of connecting outputs to realize the OR function is known as the *wired-OR* connection.

In fact, the circuit can be generalized to form a *bus* that transfers the selected source signal to the selected destination. The bus shown in (b) is simply a *common path* that is shared by all the source-to-destination data transfers. $W$ is an additional destination. In this circuit, only one source and one destination can be activated at any given time. For example, to transfer the data from $R$ to $Z$, the control signals $C_3$ and $C_5$ are activated simultaneously; similarly, $Q$ is connected to $W$ when $C_2$ and $C_6$ are activated, and so on. All the sources are wired-OR to the bus, and only one of them will be active at any given time. However, more than one destination can be activated simultaneously if the same source signal must be transferred to several destinations. To transfer $P$ to both $Z$ and $W$, control signals $C_1$, $C_5$, and $C_6$ are activated simultaneously. Buses are commonly used in digital systems when a large number of source and destinations must be interconnected.



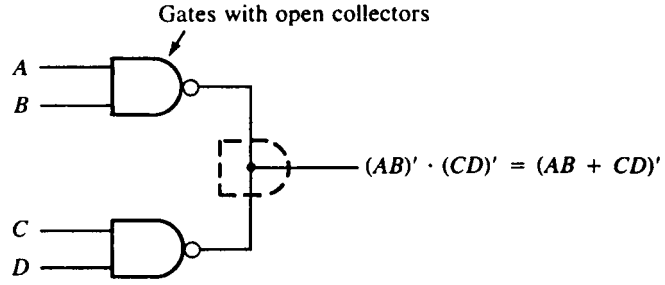

(a) A data transfer circuit (b) Bus

**Figure 1.27** Data transfer circuits

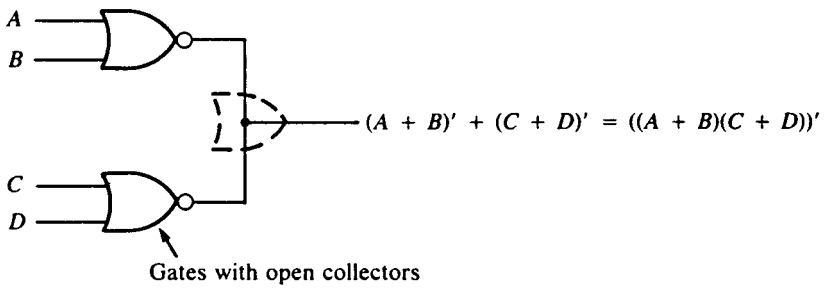
Using gates whose outputs can be connected to form the wired-OR reduces the complexity of the bus interconnection.

Two types of gates are available with special outputs that can be used in this mode: (a) gates with *open collector* (or free collector) outputs and (b) gates with *tristate outputs*.

Figure 1.28 shows two circuits. When the outputs of the TTL open collector NAND gates are tied together, an AND is realized, as shown in (a). The second level will not have a gate. This fact is illustrated with the dotted gate symbol. Note that this *wired-AND* capability results in the realization of an AND–OR–INVERT circuit (i.e., a circuit with a first level of AND gates and an OR–INVERT or NOR gate in the second level) with only one level of gates. Similarly, when open collector ECL NOR gates are used in the first level, we realize an OR when the outputs are tied together, as shown in (b). This *wired-OR* capability results in the realization of an OR–AND–INVERT circuit (i.e., a circuit with a first level




(a) TTL NANDs, WIRED-AND, AND–OR–INVERT




(b) ECL NORs, WIRED-OR, OR–AND–INVERT

**Figure 1.28** Open collector circuits

of OR gates and a second level consisting of one AND–INVERT or NAND gate) with just one level of gates.

There is a limit to the number of outputs (typically about 10 in TTL) that can be tied together. When this limit is exceeded, ICs with tristate outputs can be used in place of open collector ICs.

In addition to providing two logic levels (0 and 1), the output of these ICs can be made to stay at a *high-impedance* state. An *enable* input signal is used for this purpose. The output of the IC is either at logic 1 or 0 when it is enabled (i.e., when the enable signal is active). When it is not enabled, the output will be at the high-impedance state and is equivalent in effect to the IC not being in the circuit. It should be noted that the high impedance state is not one of the logic levels, but rather is a state in which the gate is not electrically connected to the rest of the circuit.

The outputs of tristate ICs can also be tied together to form a wired-OR as long as *only one* IC is enabled at any time. Note that in the case of a wired-OR (or wired-AND) formed using open collector gates, more than one output can be active simultaneously.

Figure 1.29 shows the bus circuit of Example 4.11 using tristate gates. The control inputs now form the enable inputs of the tristate gates. Tristate outputs allow more outputs to be connected together than the open collector ICs do.

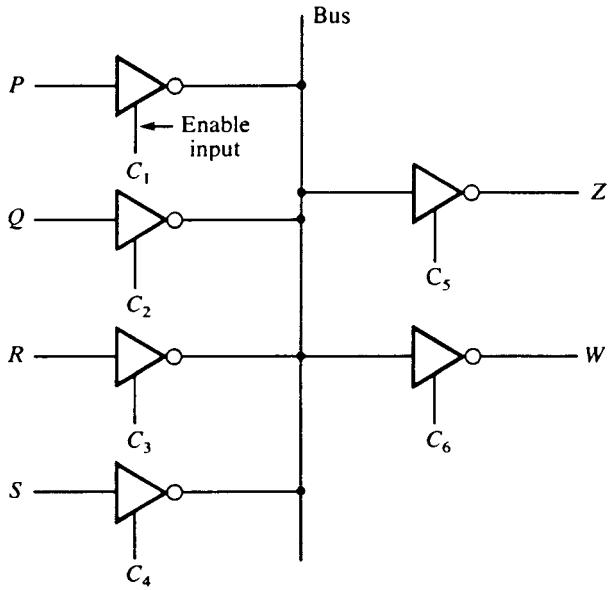



**Figure 1.29** Bus using tristate gates

Figure 1.30 shows a tristate IC (TTL 74241). This IC has eight tristate buffers, four of which are enabled by the signal on pin 1 and the other four by the signal on pin 19. Pin 1 is active-low enable, while pin 19 is active-high enable. Representative operating values are shown in (b).

ICs come with several other special features. Some ICs provide both the true and complement outputs (e.g., ECL 10107); some have a STROBE input signal that needs to be active in order for the gate output to be active (e.g., TTL 7425). Thus, STROBE is an enable input. Figure 1.31 illustrates these ICs. For further details, consult the IC manufacturer manuals listed at the end of this chapter; they provide a complete listing of the ICs and their characteristics. Refer to Appendix C for details on some popular ICs.

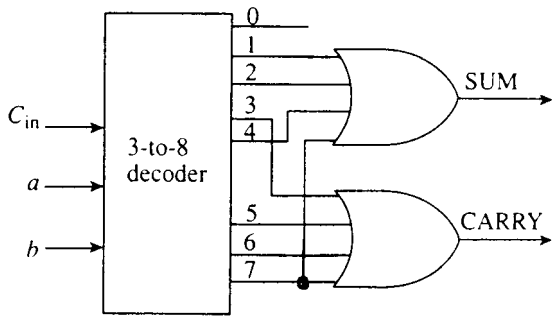
### 1.7.5 Designing with ICs

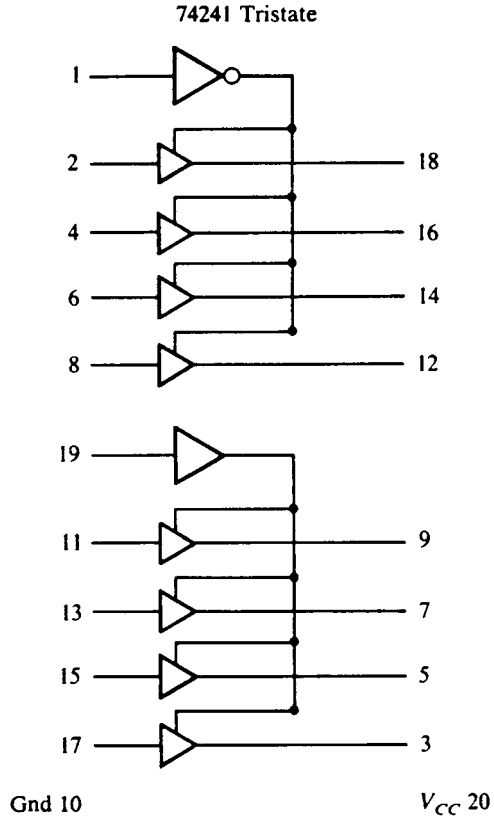
NAND–NAND or NOR–NOR implementations are extensively used in designing with ICs. NAND and NOR functions are basic to IC fabrication, and using a single type of gate in the implementation is preferable because several identical gates are available on one chip. Logic designers usually choose an available IC (decoder, adder, etc.) to implement functions rather than implement at the gate level as discussed earlier in this chapter.

Several nonconventional design approaches can be taken in designing with ICs. For example, since decoders are available as MSI components, the outputs of a decoder corresponding to the input combination where a circuit provides an output of 1 can be ORed to realize a function.

---

**Example 1.26** An implementation of a full adder using a 3-to-8 decoder (whose outputs are assumed to be high-active) is shown in the following diagram:






(a) Circuit

| Control | Input | Output | Control | Input | Output |
|---|---|---|---|---|---|
| Pin 1 | 2 | 18 | Pin 19 | 11 | 9 |
| *L* | *L* | *L* | *L* | *L* | *Z* |
| *L* | *H* | *H* | *L* | *H* | *Z* |
| *H* | *L* | *Z* | *H* | *L* | *L* |
| *H* | *H* | *Z* | *H* | *H* | *H* |

*Z* = High impedance

(b) Representative operating values

**Figure 1.30** TTL 74241 tristate buffer

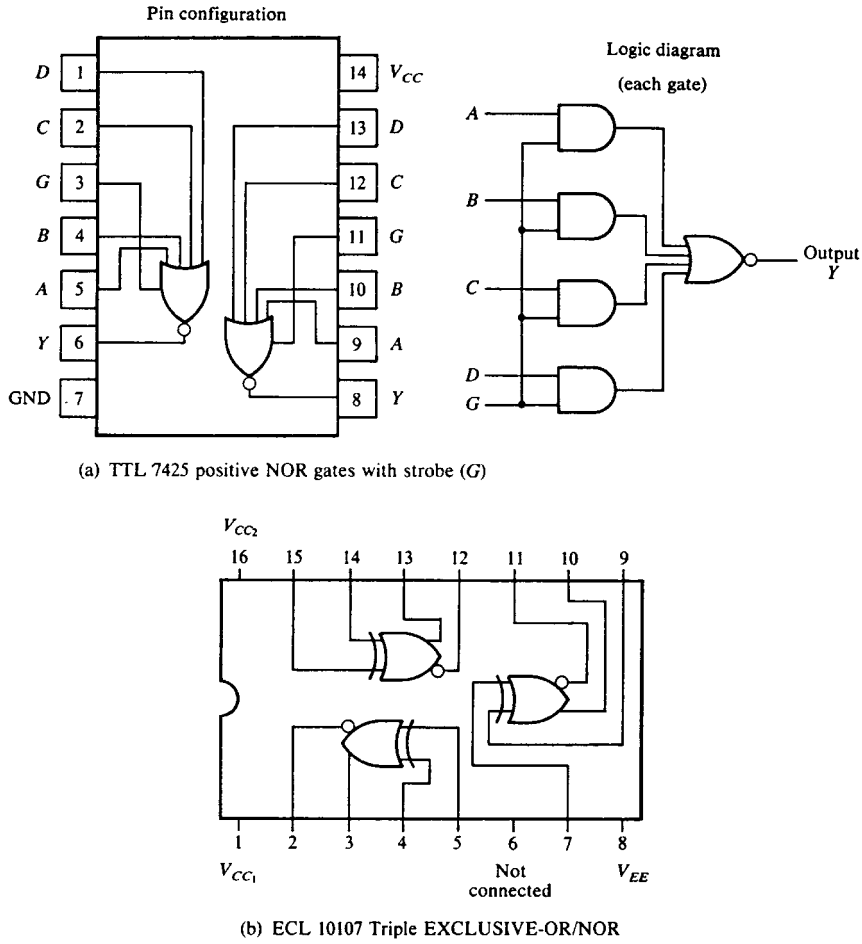
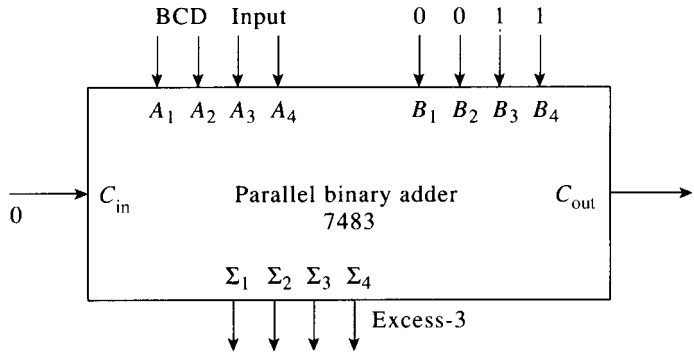



**Figure 1.31** Some special ICs

(Can you implement the full adder if the outputs of the decoder are low-active as in TTL 74155?)

---

**Example 1.27** As another example, consider the BCD to Excess-3 code converter from Fig. 1.16. This circuit can be realized using a 4-bit parallel binary adder IC (TTL 7483) as shown in the following circuit:




The *BCD* code word is connected to one of the two 4-bit inputs of the adder, and a constant of 3 (i.e., 0011) is connected to the other input. The output is the *BCD* input plus 3, which is the Excess-3 code.

---

In practice, logic designers select available MSI and LSI components first to implement as much of the circuit as possible and use SSI components as required to complete the design.

## 1.8 LOADING AND TIMING

Two main problems to be resolved in designing with ICs are *loading* and *timing*. A loading problem occurs in the event that the output of one gate can not *drive* the subsequent gates connected to it. In practice, there is a limit to the number of gate inputs that can be connected to the output of a gate. This limit is called the *fan-out* of the gate. If the fan-out limit is exceeded, the signals degrade, and hence the circuit does not perform properly. This loading problem can be solved by providing a *buffer* at the output of the loaded gate, either by using a separate inverting or noninverting buffer or by replacing the loaded gate with one that has a higher fan-out. The number of inputs to the gate is referred to as its *fan-in*.

Timing problems in general are not critical in a simple combinational circuit. However, a timing analysis is usually necessary in any complex circuit. Timing diagrams are useful in such analysis. Figure 1.32 shows the timing characteristics of a NOT gate. The $X$-axis indicates time. Logic values 1 and 0 are shown (as magnitudes of a voltage) on the $Y$-axis. Figure 1.33 shows the timing diagram for a simple combinational circuit. At $t_0$, all 3 inputs $A$, $B$, and $C$ are at 0. Hence, $Z_1$, $Z_2$, and $Z$ are all 0. At $t_1$, $B$ changes to 1. Assuming gates with no delays (ideal gates), $Z_1$ changes to 1 at $t_1$ and

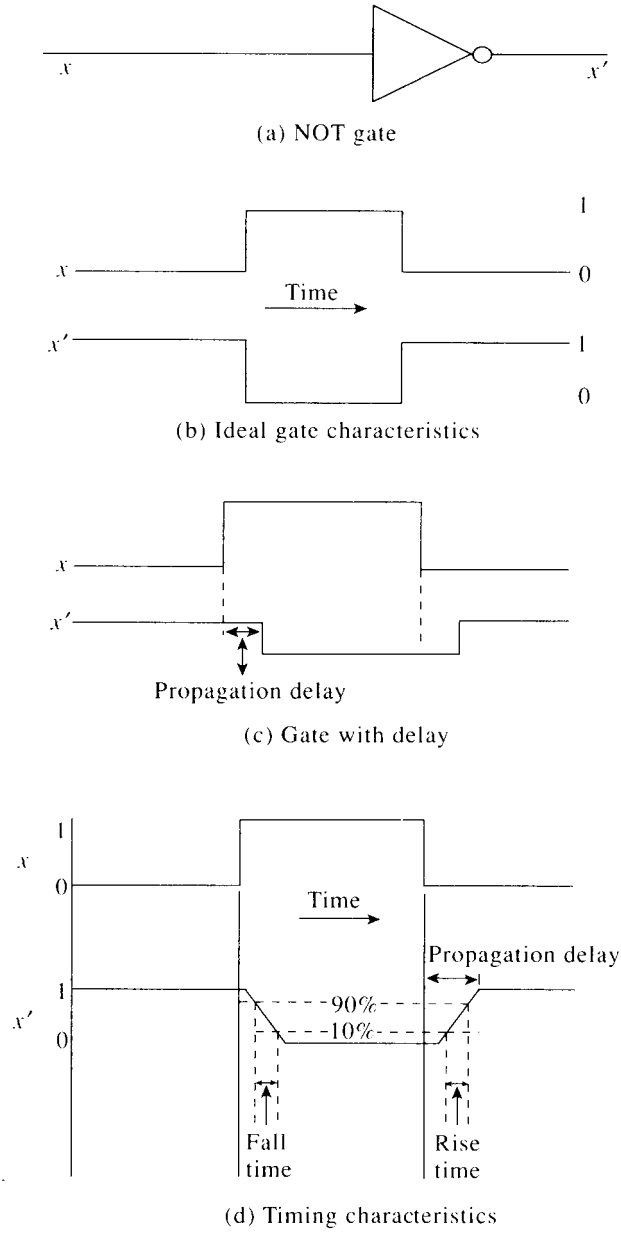



**Figure 1.32** Timing characteristics and models of an IC

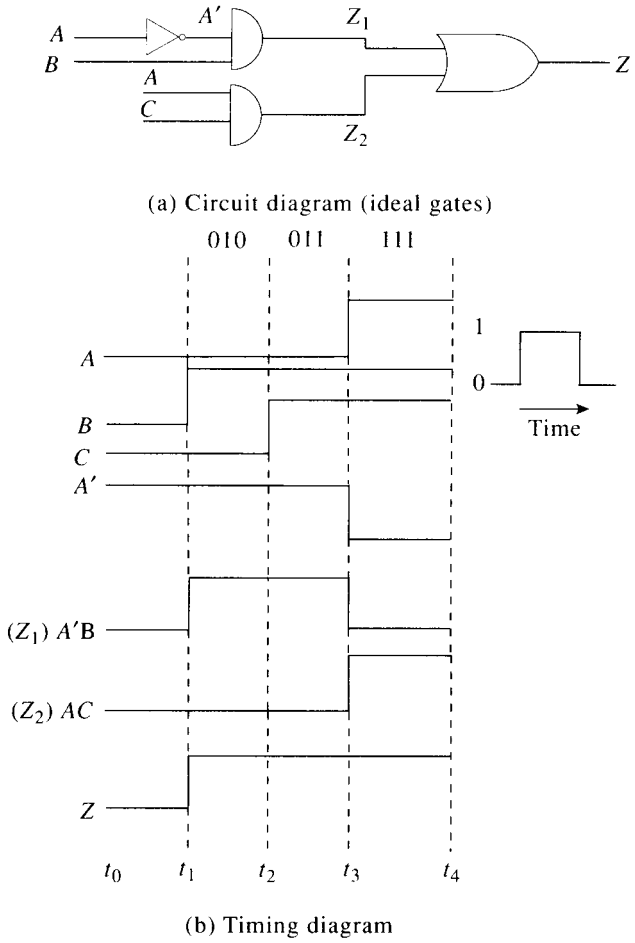
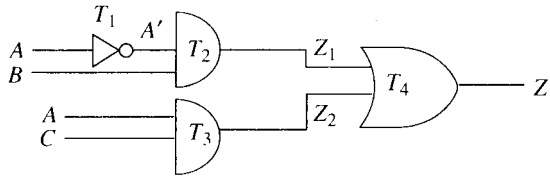


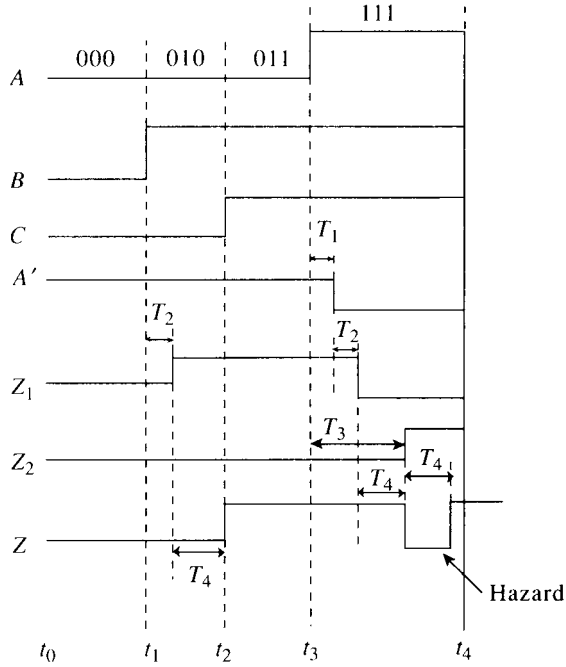

**Figure 1.33** Timing analysis of a combinational circuit

hence $Z$ also changes to 1. At $t_2$. $C$ changes to 1 resulting in no changes in $Z_1$, $Z_2$, or $Z$. At $t_3$, $A$ changes to 1, pulling $A'$ to 0; $Z_1$ to 0 and $Z_2$ to 1; $Z$ remains at 1. This timing diagram can be expanded to indicate all the other combinations of inputs. It will then be a graphical way of representing the truth table.

We can also analyze the effects of gate delays using a timing diagram. Figure 1.34 is such an analysis for the above circuit, where the gate delays are shown as $T_1$, $T_2$, $T_3$, and $T_4$. Assume that the circuit starts at $t_0$ with all the inputs at 0. At $t_1$, $B$ changes to 1. This change results in a change in $Z_1$ at




(a) Circuit diagram (gates with delays).
Assume $T_3 > (T_1 + T_2)$




(b) Timing diagram

**Figure 1.34** Timing analysis showing gate delays

$(t_1 + T_2)$, rather than at $t_1$. This change in $Z_1$ causes $Z$ to change $T_4$ later (i.e., at $t_1 + T_2 + T_4$). Changing of $C$ to 1 at $t_2$ does not change any other signal value. When $A$ is raised to 1 at $t_3$, $A'$ falls to 0 at $(t_3 + T_1)$, $Z_1$ falls to 0 at $(t_3 + T_1 + T_2)$, and $Z_2$ raises to 1 at $(t_3 + T_3)$. If $T_3 > (T_1 + T_2)$, there is a time period in which both $Z_1$ and $Z_2$ are 0, contributing a "glitch" at $Z$. $Z$ rises back to 1, $T_4$ after $Z_2$ rises to 1. This momentary transition of $Z$ to 0 might cause some problems in a complex circuit. Such *hazards* are the results of unequal delays in the signal paths of a circuit. They can be prevented by

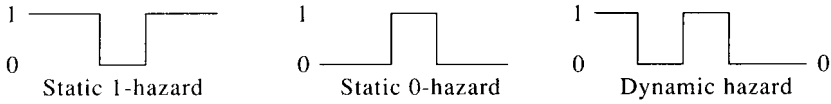



**Figure 1.35** Hazards

adding additional circuitry. This analysis indicates the utility of a timing diagram.

The hazard in the above example is referred to as *static 1-hazard*, since the output momentarily goes to 0 when it should remain at 1. This hazard occurs when the circuit is realized from the SOP form of the function. When the circuit is realized from the POS form, a *static 0-hazard* may occur, wherein the circuit momentarily gives an output of 1 when it should have remained at 0. A *dynamic hazard* causes the output to change three or more times when it should change from 1 to 0 or from 0 to 1. Figure 1.35 demonstrates the various types of hazards.

Hazards can be eliminated by including additional gates into the circuit. In general, the removal of static 1-hazards from a circuit implemented from the SOP form, also removes the static 0- and dynamic hazards. The detailed discussion of hazards is beyond the scope of this book. Refer to McCluskey (1965) and Shiva (1998) for further details on hazards.

## 1.9 SUMMARY

This chapter provided an introduction to the analysis and design of combinational logic circuits. Logic minimization procedures are discussed in Appendix B. Several intuitive methods to minimize logic circuits as used in some sections of this chapter, rather than the formal methods, will be used in the rest of the book. Readers familiar with formal methods can substitute those, to derive the minimum logic. Although the discussion on IC technology is brief, details on designing with ICs given here are sufficient to understand the information in an IC vendor's catalog and start building simple circuits. A complete understanding of the timing and loading problems helps, but is not mandatory, to understand the rest of the material in the book.

## REFERENCES

*FAST TTL Logic Series Data Handbook*, Sunnyvale, CA: Phillips Semiconductors, 1992.

Kohavi, Z. *Switching and Automata Theory*, New York, NY: McGraw-Hill, 1978.

McCluskey, E. J. *Introduction to the Theory of Switching Circuits*, New York, NY: McGraw-Hill, 1965.
Mano, M. M. *Digital Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.
*National Advanced Bipolar Logic Databook*, Santa Clara, CA: National Semiconductor Corporation, 1995.
Shiva, S. G. *Introduction to Logic Design*, New York, NY: Marcel Dekker, 1998.
*TTL Data Manual*, Sunnyvale, CA: Signetics, 1987.

## PROBLEMS

1.1 If $A = 0$, $B = 1$, $C = 0$, and $D = 1$, find the value of $F$ in each of the following:
   a. $F = AB' + C$
   b. $F = AB' + C'D + CD$
   c. $F = A'B(A + B' + C' \cdot D) + B'D$
   d. $F = (A + B')(C' + A)(A + B \cdot C)$
   e. $F = ((A + B')C + D')AB' + CD'(D' + A'(B + C'D))$.

1.2 Draw a truth table for each of the following:
   a. $Q = XY' + X'Z' + XYZ$
   b. $Q = (X' + Y)(X' + Z')(X + Z)$
   c. $Q = AB'(C' + D) + ABC' + C'D'$
   d. $Q = A'BC + AB'D' + A' + B' + CD'$
   e. $Q = (X + Y + Z')(Y' + Z)$.

1.3 State if the following identities are TRUE or FALSE.
   a. $XY' + X'Z + Y'Z = X'Y + X'Z$
   b. $(B' + C)(B' + D) = B' + CD$
   c. $A'BC + ABC' + A'BD = BD' + ABC'$
   d. $X'Z + X'Y + XZ = X'YZ' + X'YZ + X'Z$
   e. $(P + Q' + R)(P + Q' + R') = Q' + PR' + RP'$.

1.4 State if the following statements are TRUE or FALSE.
   a. $(X + Y')$ is a conjunction.
   b. $XY'Z$ is a product term.
   c. $AB'C'$ is a disjunction.
   d. $(A + B + C')$ is a sum term.
   e. $AB'$ is not included in $ABCD$.
   f. $(A + B')$ is included in $(A + B + C)$.
   g. $A + B + C'$ is included in $ABC'$.

1.5 State if the following functions are in (1) normal POS form, (2) normal SOP form, (3) canonical POS form, or (4) canonical SOP form.
   a. $F(X, Y, Z) = XY' + YZ' + Z'Y'$
   b. $F(A, B, C, D) = (A + B' + C')(A' + C' + D)(A' + C')$

c. $F(P, Q, R) = PQ' + QR'(P + Q') + (R' + Q')$
d. $F(A, B, C) = (A + B + C')(A' + B' + C')(A' + B + C')$
e. $F(A, B, C, D) = ABC'D + AB'C'D' + A'B'CD$
f. $F(A, B, C) = (A + B' + C)(A + B')(A + B + C')$
g. $F(X, Y, Z) = XY'Z + X'Y'Z + X'Y' + XYZ$
h. $F(A, B, C, D) = AB' + CD' + C'D'$.

1.6 Express each of the following functions in (1) canonical POS form and (2) canonical SOP form. (Hint: Draw the truth table for each function.)
a. $F(A, B, C) = (A + B')C' + A'C$
b. $F(X, Y, Z) = (X + Y')(X' + Z) + ZY'$
c. $F(A, B, C, D) = AB'C + A'BC'D + A'BCD' + B'D'$
d. $F(W, X, Y, Z) = WX' + Z'(Y' + W') + W'Z'Y'$.

1.7 Express F in minterm list form in each of the following:
a. $F(A, B, C) = (A + B')C' + A'C$
b. $F(X, Y, Z) = (X + Y')(X' + Z)(Z + Y')$
c. $F(P, Q, R) = \Pi M(0, 1, 5)$
d. $F(A, B, C, D) = \Pi M(1, 2, 3, 7, 9, 10, 15)$
e. $F(W, X, Y, Z) = WZ' + (W' + X')YZ' + W'Z'X'$.

1.8 Express F in maxterm list form in each of the following:
a. $F(A, B, C) = (A + B') + C' + A'C$
b. $F'(X, Y, Z) = (X + Y')(X' + Z) + ZY'$
c. $F(P, Q, R, S) = (P + Q')R' + P'S'R' + PQ'(S' + R' + Q')$
d. $F(A, B, C, D) = \Sigma m(0, 1, 5, 7, 11, 14, 15)$
e. $F(P, Q, R) = \Sigma m(2, 3, 5, 7)$.

1.9 Note that if two functions are equal, they will have the same minterms and the same maxterms. Use this facto to solve Problem 1.3.

1.10 Use only NOR gates to realize AND, OR, and NOT functions.

1.11 Given

$$F_1(A, B, C) = \underset{\substack{\uparrow \\ \text{Minterms} \\ \downarrow}}{\Sigma m(0, 1, 3)} + \underset{\substack{\uparrow \\ \text{Don't cares} \\ \downarrow}}{d(2, 7)}$$

$$F_2(A, B, C) = \Sigma m(1, 3, 5, 7) + d(6),$$

find
a. $F_1'$ in minterm list form.
b. $F_1'$ in maxterm list form.
c. $F_1' \cdot F_2$.
d. $F_1' + F_2$.

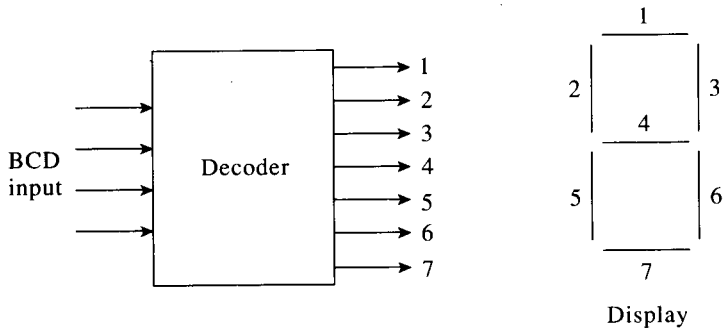
1.12 a. Design a combinational circuit with 3 inputs that produces an output that is 2s complement of the input. Refer to Appendix A for details on 2s complement system. Treat all input bits as magnitude bits, assuming there is no sign bit.

b. Extend the design in (a) to include a sign bit. That is, the output should be the 2s complement of the input only if the sign bit is 1; otherwise, it is the same as the input.

1.13 Design a comparator circuit with two 2-bit numbers $A$ and $B$ as inputs and three outputs indicating $A = B$, $A > B$, and $A < B$ conditions.

1.14 TTL7485 is a magnitude comparator. Compare the circuit diagram of 7485 from the IC catalog to your design in Problem 1.13.

1.15 Design a BCD to 7-segment decoder, with its input as the 4-bit BCD. There are seven outputs, each of which drives one segment of a 7-segment display shown below to form the decimal digit corresponding to the input code. Assume that the display device requires a 1 to turn a segment on and a 0 to turn it off.




1.16 Compare your design in Problem 1.15 with a BCD to 7-segment decoder available as an IC.

1.17 Build a full adder using two half-adders and other necessary gates.

1.18 Design a two-level NAND circuit that can multiply two 2-bit numbers and produce the 4-bit product.

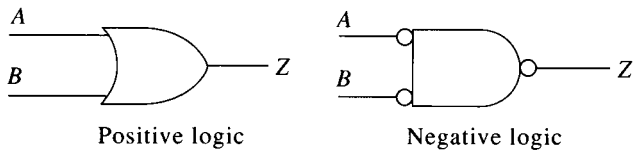
1.19 Two 2-bit numbers are to be added. Draw a truth table and design and AND-OR circuit. Compare the complexity of this circuit with the one built using a full adder and a half-adder. Which of these two circuits is faster?

1.20 Design a half-subtractor that has 2 inputs and 2 outputs: difference and borrow.

1.21 A full subtractor has 3 inputs: the 2 bits to be subtracted and a borrow-in. It produces a difference output and a borrow-output. Design a NAND-NAND circuit.

1.22 Note from the following table that the positive-logic OR gate performs as an AND gate in negative logic.

| A | B | Z |
|---|---|---|
| L | L | L |
| L | H | H |
| H | L | H |
| H | H | H |




Positive logic: $H = 1$, $L = 0$
Negative logic: $H = 0$, $L = 1$

Find the equivalents of OR, NAND, NOR, and XOR gates in negative logic.

1.23 Design a 2-to-4 decoder using only NAND gates. Also include a low-active ENABLE input to the decoder. When the ENABLE input is active, all the outputs of the decoder should be HIGH. When the ENABLE signal is active, the selected output show be LOW.

1.24 Show that the 2-to-4 decoder of Problem 1.23 can be used as a 1-to-4 demultiplexer, with ENABLE as the input and the other two inputs as the control signals.

1.25 Derive the exclusive-OR function using a 2-to-1 multiplexer.

1.26 Show that an $n$-input, one output function can be implemented using one $2^n$-to-1 multiplexer. (HINT: Each input of the multiplexer corresponds to an input combination.)

1.27 Design an adder/subtractor for two 4-bit numbers using a TTL7483 and other required gates. Assume that $C$ is a control signal: the circuit should add the two numbers when $C$ is 0; subtract them when $C$ is 1.

1.28 Design a circuit that can multiply two 4-bit numbers and produce the 8-bit product. Use 4-bit parallel adder ICs and appropriate number of AND gates in your design. AND gates are used to generate the partial products, and the adders accumulate them to obtain the final product.

1.29 Design a 5-to-32 decoder using 3-to-8 decoders.

1.30 Compare the complexity of your design in Problem 1.29 to the design of the decoder using NAND gates only.

# 2
# Synchronous Sequential Circuits

The digital circuits we have examined so far do not possess any memory. That is, the output of the combinational circuit at any time is a function of the inputs at that time. In practice, most digital systems contain memory elements in addition to the combinational logic portion, thus making them *sequential* circuits. The output of a sequential circuit at any time is a function of its external inputs and the internal *state* at that time. The state of the circuit is defined by the contents of the memory elements in the circuit and is a function of the previous states and the inputs to the circuit.

Figure 2.1 shows a block diagram of a sequential circuit with $m$ inputs, $n$ outputs, and $p$ internal memory elements. The output of the $p$ memory elements combined constitutes the state of the circuit at time $t$ (i.e., the *present state*). The combinational logic determines the output of the circuit at time $t$ and provides the next-state information to the memory elements based on the external inputs and the present state. Based on the next-state information at $t$, the contents of all the memory elements change to the next state, which is the state at time $(t + \Delta t)$. where $\Delta t$ is a time increment sufficient for the memory elements to make the transition. We will denote $(t + \Delta t)$ as $(t + 1)$ in this chapter.

There are two types of sequential circuits: *synchronous* and *asynchronous*. The behavior of a synchronous circuit depends on the signal values at discrete points of time. The behavior of an asynchronous circuit depends on the order in which the input signals change, and these changes can occur at any time.

The discrete time instants in a synchronous circuit are determined by a controlling signal, usually called a *clock*. A clock signal makes 0 to 1 and 1 to 0 transitions at regular intervals. Figure 2.2 shows two clock signals (one is the complement of the other), along with the various terms used to describe the clock. A pair of 0-to-1 and 1-to-0 transitions constitutes a *pulse*. That is, a pulse consists of a *rising edge* and a *falling edge*. The time

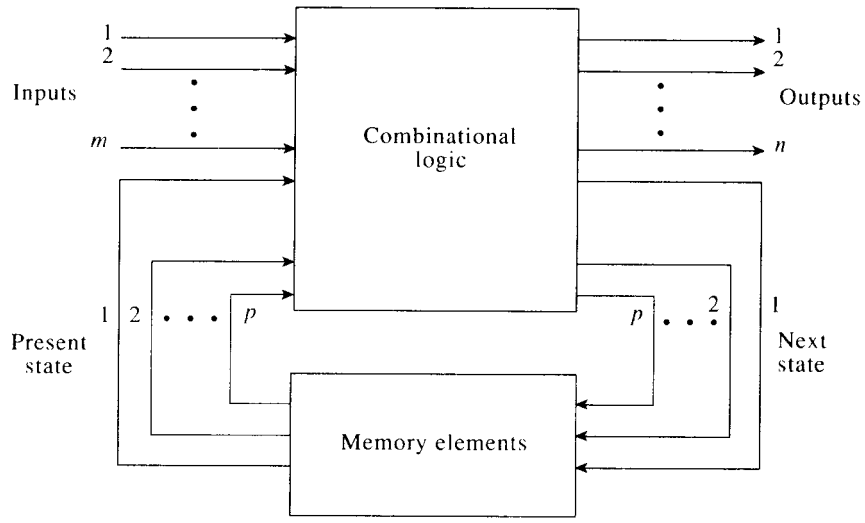



**Figure 2.1** Block diagram of a sequential circuit

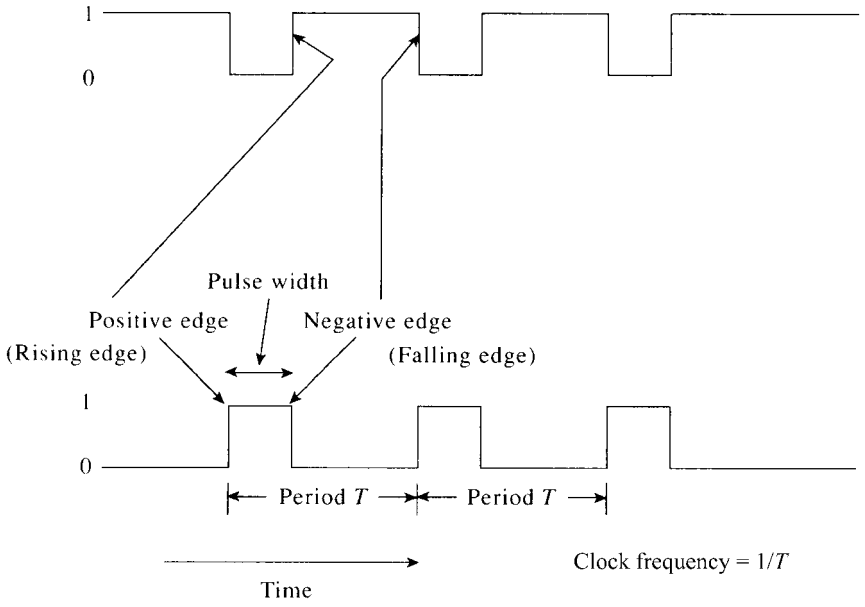



**Figure 2.2** Clock

between these transitions (edges) is the *pulse width*. The *period (T)* of the clock is the time between two correspondings edges of the clock, and the *clock frequency* is the reciprocal of its period. Although the clock in Fig. 2.2 is shown with a regular period $T$, the intervals between two pulses do not need to be equal.

Synchronous sequential circuits use flip-flops as memory elements. A flip-flop is an electronic device that can store either a 0 or a 1. That is, a flip-flop can stay in one of the two logic states, and a change in the inputs to the flip-flop is needed to bring about a change of state. Typically, there will be two outputs from a flip-flop: one corresponds to the normal state ($Q$) and the other corresponds to the complement state ($Q'$). We will examine four popular flip-flops in this chapter.

Asynchronous circuits use time-delay elements (*delay lines*) as memory elements. The delay line shown in Fig. 2.3(a) introduces a propagation delay ($\Delta t$) into its input signal. As shown in (b), the output signal is the same as

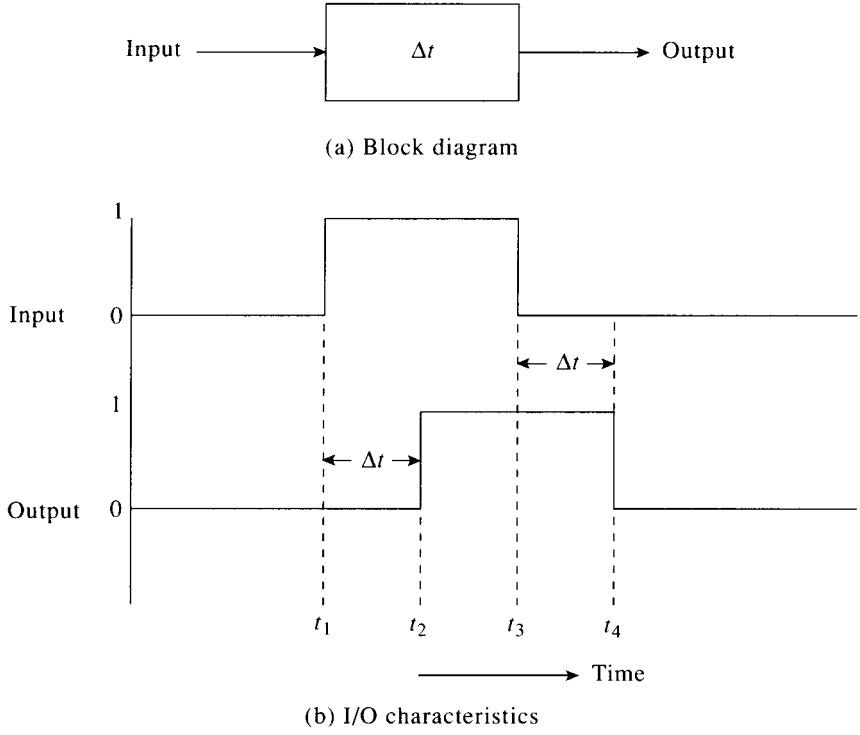



**Figure 2.3** Delay element

the input signal, except that it is delayed by $\Delta t$. For instance, the 0-to-1 transition of the input at $t_1$ occurs on the output at $t_2$, $\Delta t$ later. Thus, if delay lines are used as memory elements, the present-state information at time $t$ forms their input and the next state is achieved at $(t + \Delta t)$. In practice, the propagation delays introduced by the combinational circuit's logic gates may be sufficient to produce the needed delay, thereby not necessitating a physical time-delay element. In such cases, the model of Fig. 2.1 reduces to a combinational circuit with *feedback* (i.e., a circuit whose outputs are fed back as inputs). Thus, an asynchronous circuit may be treated as a combinational circuit with *feedback*. Because of the feedback, the changes occurring in the output as a result of input changes may in turn contribute to further changes in inputs – and the cycle of changes may continue to make the circuit unstable if the circuit is not properly designed. In general, asynchronous circuits are difficult to analyze and design. If properly designed, however, they tend to be faster than synchronous circuits.

A synchronous sequential circuit generally is controlled by pulses from a *master clock*. The flip-flops in the circuit make a transition to the new state only when a clock pulse is present at their inputs. In the absence of a single master clock, the operation of the circuit becomes unreliable, since two clock pulses arriving from different sources at the inputs of the flip-flops cannot be guaranteed to arrive at the same time (because of unequal path delays). This phenomenon is called *clock skewing*. Clock skewing can be avoided by analyzing the delay in each path from the clock source and inserting additional gates in paths with shorter delays to make the delays of all paths equal.

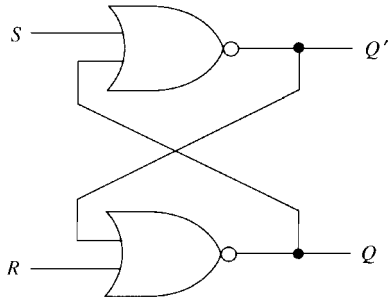
In this chapter we will describe the analysis and design procedures for synchronous sequential circuits. Refer to the books listed in the references section of this chapter for details on asynchronous circuits.

## 2.1 FLIP-FLOPS

As mentioned earlier, a flip-flop is a device that can store either a 0 or a 1. When the flip-flop contains a 1, it is said to be *set* (i.e., $Q = 1$, $Q' = 0$) and when it contains a 0 it is *reset* (i.e., $Q = 0$, $Q' = 1$). We will introduce the logic properties of four popular types of flip-flops in this section.

### 2.1.1 Set-Reset (*SR*) Flip-Flop

An *SR* flip-flop has two inputs: *S* for setting and *R* for resetting the flip-flop. An ideal *SR* flip-flop can be built using a cross-coupled NOR circuit, as shown in Fig. 2.4(a). The operation of this circuit is illustrated in (b). When




(a) Logic diagram

| $S(t)$ | $R(t)$ | $Q(t)$ | $Q'(t)$ | $Q(t+1)$ | $Q'(t+1)$ |
|---|---|---|---|---|---|
| 1 | 0 | – | – | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | – | – | 0 | 0 |

(b) Partial truth table

| Present state $Q(t)$ \ Inputs $S(t)$ $R(t)$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | – |
| 1 | 1 | 0 | 1 | – |
| | No change | Reset | Set | Not allowed |

$Q(t+1)$

(c) State table

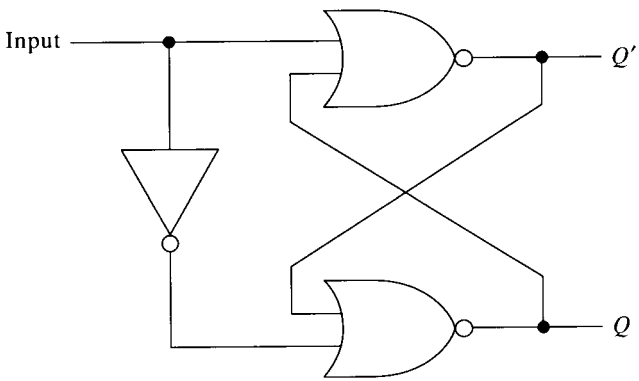
**Figure 2.4** *SR* flip-flop

inputs $S = 1$ and $R = 0$ are applied at any time $t$, $Q'$ assumes a value of 0 (one gate delay later). Since $Q'$ and $R$ are both at 0, $Q$ assumes a value of 1 (another gate delay later). Thus, in two gate delay times the circuit settles at the set state. We will denote the two gate delay times as $\Delta t$. Hence, the state at time $(t + \Delta t)$ or $(t + 1)$, designated at $Q(t + 1)$ is 1. If $S$ is changed to 0, as shown in the second row of (b), an analysis of the circuit indicates that the $Q$ and $Q'$ values do not change. If $R$ is then changed to 1, the output values change to $Q = 0$ and $Q' = 1$. Changing $R$ back to 0 does not alter the output values. When $S = 1$ and $R = 1$ are applied, both outputs assume a value of 0, regardless of the previous state of the circuit. This condition is not desirable, since the flip-flop operation requires that one output always be the complement of the other. Further, if now the input condition changes to $S = 0$ and $R = 0$, the state of the circuit depends on the order in which the inputs change from 1 to 0. If $S$ changes faster than $R$, the circuit attains the reset state; otherwise, it attains the set state.

Thus, the cross-coupled NOR gate circuit forms an *SR* flip-flop. The input condition $S = 1$ and $R = 0$ *sets* the flip-flop; the condition $S = 0$ and $R = 1$ *resets* the flip-flop. $S = 0$ and $R = 0$ constitute a "no change" condition. (The input condition $S = 1$ and $R = 1$ is not permitted to occur on the inputs.)

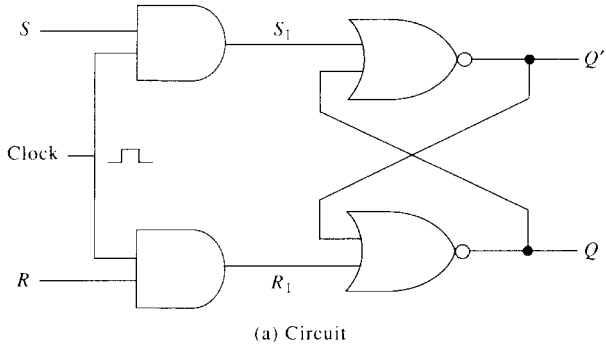
The transitions of the flip-flop from the present state $Q(t)$ to the next state $Q(t + 1)$ for various input combinations are summarized in Fig. 2.4(c). This table is called the *state table*. It has four columns (one corresponding to each input combination) and two rows (one corresponding to each state the flip-flop can be in).

Recall that the outputs of the cross-coupled NOR gate circuit in Fig. 2.4 do not change instantaneously once there is a change in the input condition. The change occurs after a delay of $\Delta t$, which is the equivalent of at least two gate delays. This is an asynchronous circuit, since the outputs change as the inputs change. The circuit is also called the *SR latch*. As the name implies, such a device is used to *latch* (i.e., store) the data for later use. In the following circuit, the data (1 or a 0) on the INPUT line are latched by the flip-flop:




A clock input can be added to the circuit of Fig. 2.4 to construct a *clocked SR flip-flop*, as shown in Fig. 2.5(a). As long as the clock stays at 0 (i.e., in the absence of the clock pulse), the outputs of the two AND gates ($S_1$ and $R_1$) are 0, and hence the state of the flip-flop does not change. The $S$ and $R$ values are impressed on the flip-flop inputs ($S_1$ and $R_1$) only during the clock pulse. Thus, the clock controls all the transitions of this synchronous circuit. The graphic symbol for the clocked *SR* flip-flop is shown in (b).

Given the present state and the $S$ and $R$ input conditions, the next state of the flip-flop can be determined, as shown in the *characteristic table*




(a) Circuit

(b) Graphic symbol

| $Q(t)$ | $SR$ | $Q(t+1)$ |
|---|---|---|
| 0 | 00 | 0 |
| 0 | 01 | 0 |
| 0 | 10 | 1 |
| 0 | 11 | – |
| 1 | 00 | 1 |
| 1 | 01 | 0 |
| 1 | 10 | 1 |
| 1 | 11 | – |

(c) Characteristic table

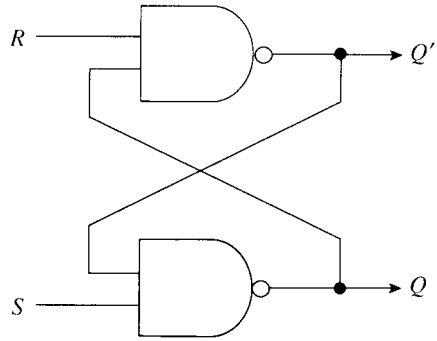
Note that the characteristic table is the state table rearranged into the form of a truth table.

**Figure 2.5** Clocked *SR* flip-flop

in (c). This table is obtained by rearranging the state table in Fig. 2.4(c) so that the next state can be determined easily once the present state and the input condition are known.

Figure 2.6 shows an *SR* flip-flop formed by cross-coupling two NAND gates along with the truth table. As can be seen by the truth table, this circuit requires a 0 on its input to change the state, unlike the circuit of Fig. 2.4, which required a 1.

As mentioned earlier, it takes at least two gate delay times for the state transition to occur after there has been a change in the input condition of the flip-flop. Thus the pulse width of the clock controlling the flip-flop must be at least equal to this delay, and the inputs should not change until the transition is complete. If the pulse width is longer than the delay, the state transitions resulting from the first input condition change are overridden by any subsequent changes in the inputs during the clock pulse. If it is necessary to recognize all the changes in the input conditions, however, the pulse width must be short enough. The pulse width and the clock frequency thus must be adjusted to accommodate the flip-flop circuit transition time




(a) Circuit

| $S(t)$ | $R(t)$ | $Q(t)$ | $Q'(t)$ | $Q(t+1)$ | $Q'(t+1)$ | |
|---|---|---|---|---|---|---|
| 0 | 1 | – | – | 1 | 0 | Set |
| 1 | 1 | 1 | 0 | 1 | 0 | No change |
| 1 | 0 | 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 0 | 1 | 0 | 1 | No change |
| 0 | 0 | – | – | 1 | 1 | Not allowed |

(b) Truth table

**Figure 2.6** *SR* flip-flop formed by cross-coupled NAND gates

and the rate of input change. (The timing characteristics and requirements of flip-flops are further discussed later in this chapter.)

Figure 2.7 shows the graphic symbol for an *SR* flip-flop with both clocked (*S* and *R*) and asynchronous inputs (*preset* and *clear*). A clock is not required to activate the flip-flop through the asynchronous inputs. Asynchronous (or direct) inputs are not used during the regular operation of the flip-flop. They generally are used to initialize the flip-flop to either the set or the reset state. For instance, when the circuit power is turned on, the state of the flip-flops cannot be determined. The direct inputs are used to initialize the state, either manually through a "master clear" switch or through a power-up circuit that pulses the direct input of all the flip-flops in the circuit.

We will now examine three other commonly used flip-flops. The preset, clear, and clocked configurations discussed above apply to these flip-flops as well. In the remaining portions of this chapter, if a reference to a signal does not show a time associated with it, it is assumed to be the current time *t*.

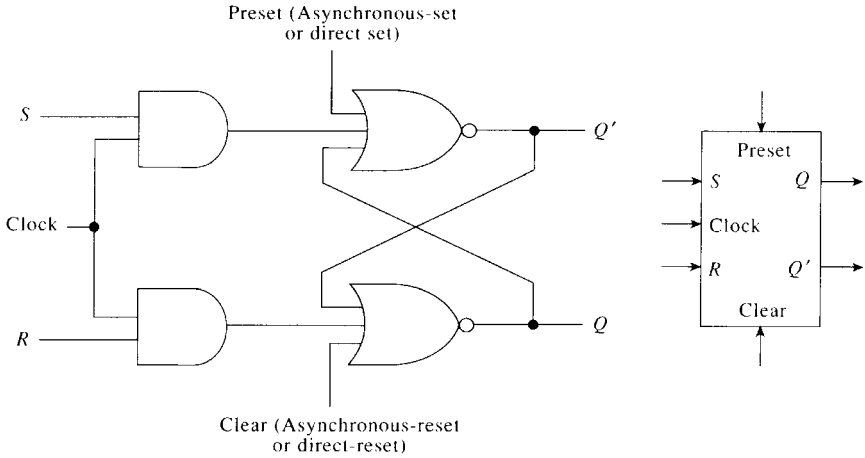



**Figure 2.7** A clocked *SR* flip-flop with preset/clear

### 2.1.2 *D* Flip-Flops

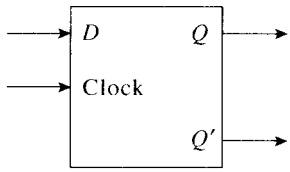
Figure 2.8 shows a *D* (delay or data) flip-flop and its state table. The *D* flip-flop assumes the state of the *D* input; that is, $Q(t+1) = 1$ if $D(t) = 1$, and $Q(t+1) = 0$ if $D(t) = 0$. The function of this flip-flop is to introduce a unit delay ($\Delta t$) in the signal input at *D*. Hence this flip-flop is known as a *delay flip-flop*. It is also called a *data flip-flop*, since it stores the data on the *D* input line.

The *D* flip-flop is a modified *SR* flip-flop that is obtained by connecting *D* to an *S* input and $D'$ to an *R* input, as shown in (c). A clocked *D* flip-flop is also called a *gated D-latch*, in which the clock signal gates the data into the latch.

The next state of the *D* flip-flop is the same as the data input at any time, regardless of the present state. This is illustrated by the characteristic table shown in (d).

### 2.1.3 *JK* Flip-Flops

The *JK* flip-flop is a modified *SR* flip-flop in that the $J = 1$ and $K = 1$ input combination is allowed to occur. When this combination occurs, the flip-flop complements its state. The *J* input corresponds to the *S* input, and the *K* input corresponds to the *R* input of an *SR* flip-flop. Figure 2.9 shows the graphic symbol, the state table, the characteristic table, and the realization

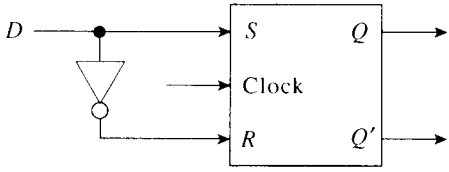



(a) Graphic symbol

| $Q(t)$ \ $D(t)$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |

$Q(t+1)$

(b) State table

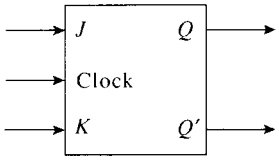



(c) Flip-flop realized from an *SR* flip-flop

| $Q(t)$ | $D$ | $Q(t+1)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(d) Characteristic table

**Figure 2.8** *D* flip-flop




(a) Graphic symbol

| $Q(t)$ \ $JK(t)$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |

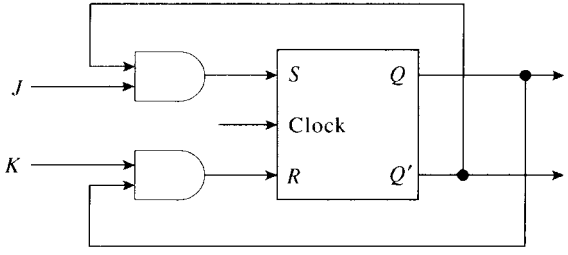
No change Reset Set Complement

$Q(t+1)$

(b) State table

| $Q(t)$ | $JK$ | $Q(t+1)$ |
|---|---|---|
| 0 | 00 | 0 |
| 0 | 01 | 0 |
| 0 | 10 | 1 |
| 0 | 11 | 1 |
| 1 | 00 | 1 |
| 1 | 01 | 0 |
| 1 | 10 | 1 |
| 1 | 11 | 0 |

(c) Characteristic table




(d) Realization of *JK* flip-flop using an *SR* flip-flop

**Figure 2.9** *JK* flip-flop

of a *JK* flip-flop using an *SR* flip-flop. (See Problem 2.1 for a hint on how to convert one type of flip-flop into another.)

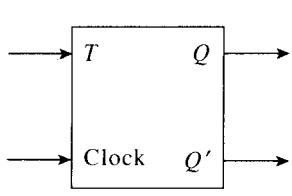
### 2.1.4 *T* Flip-Flops

Figure 2.10 shows the graphic symbol, state table, and the characteristic table for a *T* (toggle) flip-flop. This flip-flop complements its state when $T = 1$ and remains in the same state as it was when $T = 0$. A *T* flip-flop can be realized by connecting the *J* and *K* inputs of a *JK* flip-flop as shown in Fig. 2.10(d).

### 2.1.5 Characteristic and Excitation Tables

The characteristic table of a flip-flop is useful in the analysis of sequential circuits, since it provides the next-state information as a function of the present state and the inputs. The characteristics tables of all the flip-flops are given in Fig. 2.11 for ready reference.

The *excitation tables* (or the *input tables*) shown in Fig. 2.12 for each flip-flop are useful in designing sequential circuits, since they describe the excitation (or input condition) required to bring the state transition of the flip-flop from $Q(t)$ to $Q(t + 1)$. These tables are derived from the state tables




(a) Graphic symbol

| $Q(t)$ | $T(t)$ = 0 | $T(t)$ = 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

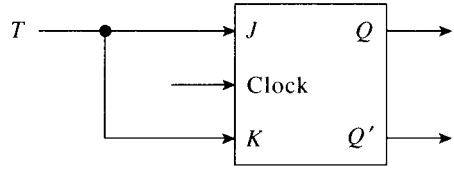
No change  Complement

$Q(t + 1)$

(b) State table

| $Q(t)$ | $T$ | $Q(t + 1)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(c) Characteristic table




(d) *T* Flip-flop realized from a *JK* flip-flop

**Figure 2.10** *T* flip-flop

| $Q(t)$ | $SR$ | $Q(t+1)$ |
|---|---|---|
| 0 | 00 | 0 |
| 0 | 01 | 0 |
| 0 | 10 | 1 |
| 0 | 11 | - |
| 1 | 00 | 1 |
| 1 | 01 | 0 |
| 1 | 10 | 1 |
| 1 | 11 | - |

(a) $SR$ flip-flop

| $Q(t)$ | $D$ | $Q(t+1)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(b) $D$ flip-flop

| $Q(t)$ | $JK$ | $Q(t+1)$ |
|---|---|---|
| 0 | 00 | 0 |
| 0 | 01 | 0 |
| 0 | 10 | 1 |
| 0 | 11 | 1 |
| 1 | 00 | 1 |
| 1 | 01 | 0 |
| 1 | 10 | 1 |
| 1 | 11 | 0 |

(c) $JK$ flip-flop

| $Q(t)$ | $T$ | $Q(t+1)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(d) $T$ flip-flop

**Figure 2.11** Characteristic tables

of the corresponding flip-flops. Consider the state table for the $SR$ flip-flop shown in Fig. 2.4. For a transition of the flip-flop from state 0 to 0 (as shown by the first row of the state table), the input can be either $SR = 00$ or 01. That is, an $SR$ flip-flop makes a transition from 0 to 0 as long as $S$ is 0 and $R$ is either 1 or 0. This excitation requirement is shown as $SR = 0d$ in the first row of the excitation table. A transition from 0 to 1 requires an input of $SR = 10$; a transition from 1 to 0 requires $SR = 01$ and that from 1 to 1 requires

| $Q(t)$ | $Q(t+1)$ | $SR$ | $D$ | $JK$ | $T$ |
|---|---|---|---|---|---|
| 0 | 0 | $0d$ | 0 | $0d$ | 0 |
| 0 | 1 | 10 | 1 | $1d$ | 1 |
| 1 | 0 | 01 | 0 | $d1$ | 1 |
| 1 | 1 | $d0$ | 1 | $d0$ | 0 |

*Note:* $d$ = don't-care (0 or 1)

**Figure 2.12** Excitation tables

$SR = d0$. Thus, the excitation table accounts for all four possible transitions. The excitation tables for the other three flip-flops are similarly derived.

## 2.2 TIMING CHARACTERISTICS OF FLIP-FLOPS

Consider the cross-coupled NOR circuit forming an *SR* flip-flop. Figure 2.13 shows a timing diagram, assuming that the flip-flop is at state 0 to begin with. At $t_1$, input $S$ changes from 0 to 1. In response to this, $Q$ changes to 1 at $t_2$, a delay of $\Delta t$ after $t_1$. $\Delta t$ is the time required for the circuit to settle to the new state. At $t_3$, $S$ goes to 0, with no change in $Q$. At $t_4$, $R$ changes to 1, and hence $Q$ changes to 0, $\Delta t$ time later at $t_5$. At $t_6$, $R$ changes to 0, with no effect on $Q$.

Note that the $S$ and $R$ inputs should each remain at their new data value at least for time $\Delta t$ for the flip-flop to recognize the change in the input condition (i.e., to make the state transition). This time is called the *hold time*. Now consider the timing diagram for the clocked *SR* flip-flop shown in Fig. 2.14. The clock pulse width $w = t_8 - t_1$. $S$ changes to 1 at $t_2$, and in response to it $Q$ changes to 1 at $t_3$, $\Delta t$ later. Since the clock pulse is still at 1 when $R$ changes to 1 at $t_5$, $Q$ changes to 0 at $t_6$. If the pulse width

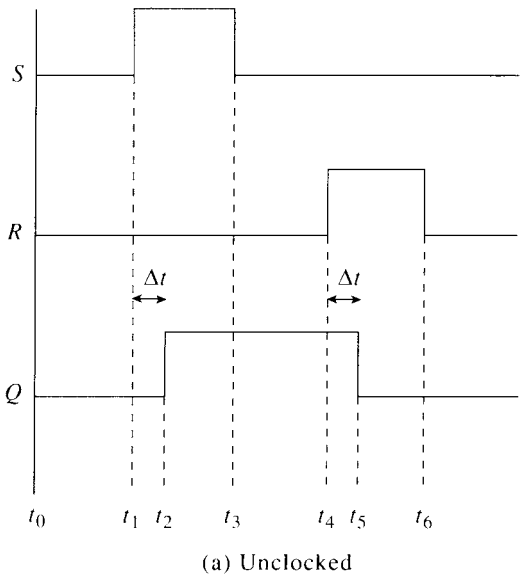



**Figure 2.13** Timing diagram for an *SR* flip-flop (unclocked)

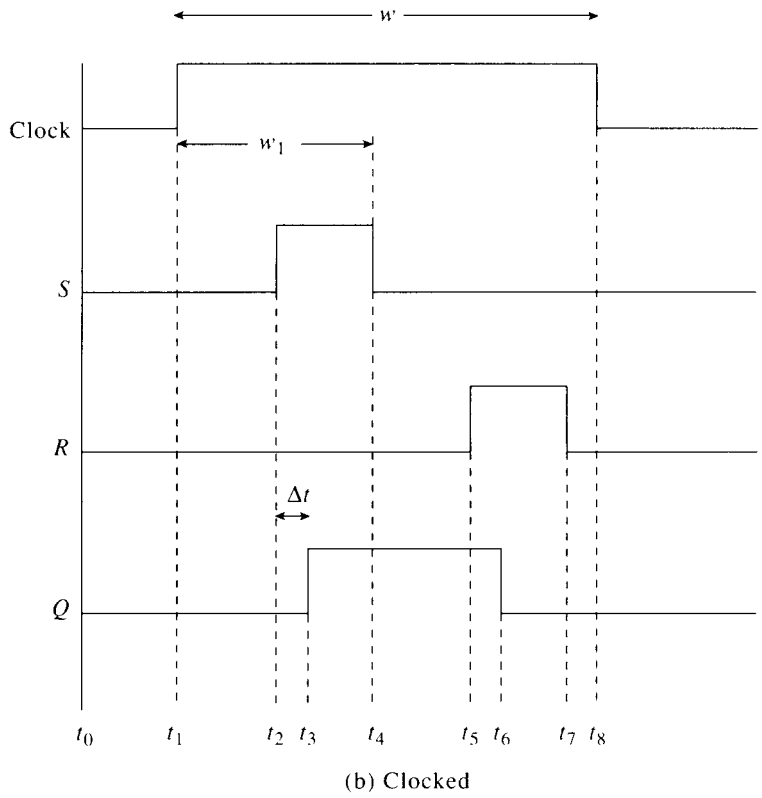



**Figure 2.14** Timing diagram for an *SR* flip-flop (clocked)

were to be $w_1 = t_4 - t_1$, only the change in $S$ would have been recognized. Thus, in the case of a clocked *SR* flip-flop, the clock pulse width should at least equal $\Delta t$ for the flip-flop to change its state in response to a change in the input. If the pulse width is greater than $\Delta t$, and $S$ and $R$ values should change no more than once during the clock pulse, since the flip-flop circuit will keep changing states as a result of each input change and registers only the last input change. As such, the clock pulse width is a critical parameter for the proper operation of the flip-flop.

Consider the timing diagram of Fig. 2.15 for a $T$ flip-flop. When $T$ changes to 1 at $t_1$, the flip-flop changes from its original state of 0 at $t_2$, $\Delta t$ time later. Since the $T$ flip-flop circuit contains a feedback path from its outputs to its input, if the $T$ input stays at 1 longer (i.e., beyond $t_2$), the output would be fed back to the input and the flip-flop changes state again. To avoid this oscillation, $w$ must always be less than $\Delta t$.

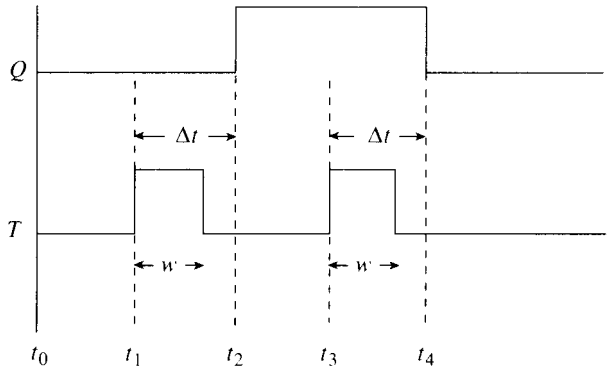



**Figure 2.15** Timing diagram for a *T* flip-flop

In order to avoid such problems resulting from clock pulse width, flip-flops in practice are designed either as *master-slave flip-flops* or as *edge-triggered flip-flops*, which are described next.

### 2.2.1 Master–Slave Flip-Flops

The master–slave configuration is shown in Fig. 2.16(a). Here, two flip-flops are used. The clock controls the separation and connection of the circuit inputs from the inputs of the master, and the inverted clock controls the separation and connection of slave inputs from the master outputs. In practice, the clock signal takes a certain amount of time to make the transition from 0 to 1 and 1 to 0, as shown by $t_R$ and $t_F$, respectively, in the timing diagram (b). As the clock changes from 0 to 1, at point A the slave stage is disconnected from the master stage; at point B, the master is connected to the circuit inputs and changes its state based on the inputs. At point C, as the clock makes it transition from 1 to 0, the master stage is isolated from the inputs; and at D, the slave inputs are connected to the outputs of the master stage. The slave flip-flop changes its state based on its inputs, and the slave stage is isolated from the master stage at A again. Thus, the master–slave configuration results in at most one state change during each clock period, thereby avoiding the race conditions resulting from clock pulse width.

Note that the inputs to the master stage can change after the clock pulse while the slave stage is changing its state without affecting the operation of the master–slave flip-flop, since these changes are not recognized by the master until the next clock pulse. Master–slave flip-flops are especially useful when the input of a flip-flop is a function of its own output.

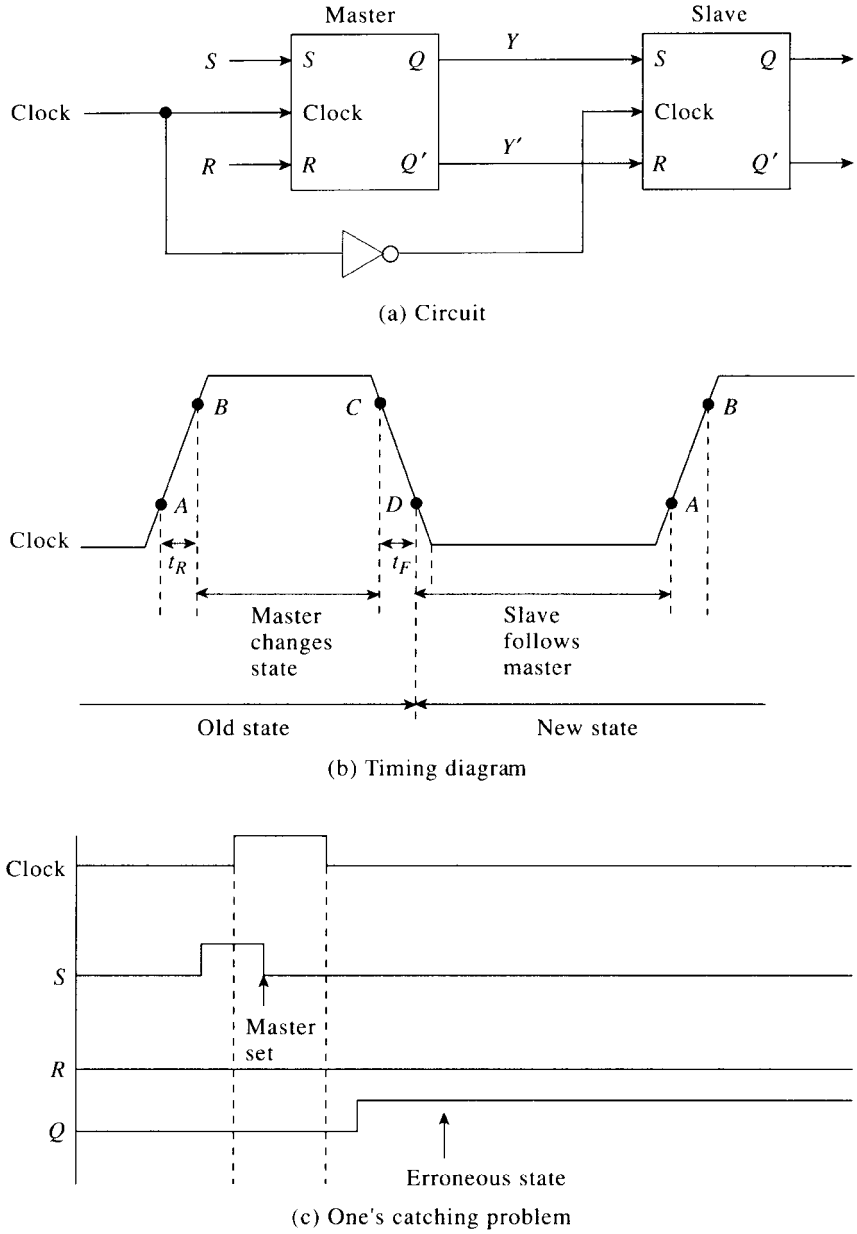



**Figure 2.16** Master–slave flip-flop

Consider the timing diagram of Fig. 2.16(c) for a master–slave flip-flop. Here, $S$ and $R$ initially are both 0. The flip-flop should not change its state during the clock pulse. However, a glitch in the $S$ line while clock is high sets the master stage, which in turn is transferred to the slave stage, resulting in an erroneous state. This is called *one's catching* problem and can be avoided by ensuring that all the input changes are complete and the inputs stable well before the leading edge of the clock. This timing requirement is known as the *setup* time ($t_{\text{setup}}$). That is, $t_{\text{setup}} > w$, the clock pulse width. This can be achieved either by a narrow clock pulse width (which is difficult to guarantee) or by a large setup time (which reduces the flip-flop's operating speed). Edge-triggered flip-flops are preferred over master–slave flip-flops because of the one's catching problem associated with the latter.

### 2.2.2 Edge-Triggered Flip-Flops

Edge-triggered flip-flops are designed so that they change their state based on input conditions at either the rising or the falling edge of the clock. The rising edge of the clock triggers a positive edge-triggered flip-flop (as shown in Fig. 2.14), and the falling edge of the clock triggers a negative edge-triggered flip-flop. Any change in input values after the occurrence of the triggering edge will not bring about a state transition in these flip-flops until the next triggering edge.

Figure 2.17(a) shows the most common trailing edge-triggered flip-flop circuit, built out of three cross-coupled NOR flip-flops. Flip-flops 1 and 2 serve to set the inputs to the third flip-flop at appropriate values based on the clock and $D$ inputs. Consider the clock and $D$ input transitions shown in (b). Flip-flop 3 is reset initially (i.e., $Q = 0$). When the clock goes to 1 at $t_0$, point $W$ goes to 0 (after one gate delay). Since $Z$ remains at 0, flip-flop 3 does not change its state. While the clock pulse is at 1, $X$ and $Y$ follow $D$ (i.e., $X = D'$; and $Y = D$), as at $t_1$. When the clock changes to 0 at $t_2$, $Z$ changes to 1 (after a delay) at $t_3$, but $W$ remains 0. Consequently, flip-flop 3 changes its state to 1 (after a delay). Thus the state change is brought about by the trailing edge of the clock.

While the clock is at 0, change in the $D$ input does not change either $Z$ or $W$, as shown at $t_4$ and $t_5$.

$Z$ is 1 and $W$ is 0 at $t_6$ when the clock changes to 1 and $Z$ goes to 0. At $t_7$, the $D$ input changes to 0. Since the clock is at 1, $X$ and $Y$ change accordingly (after a delay). These changes result in changing $W$ to 1 at the trailing edge of the clock at $t_8$. Since $Z = 0$ and $W = 1$, flip-flop 3 changes to 0.

As can be seen by the timing diagram shown in (b), after the trailing edge of the clock pulse, either $W$ or $Z$ becomes 1. When $Z$ is 1, D is blocked

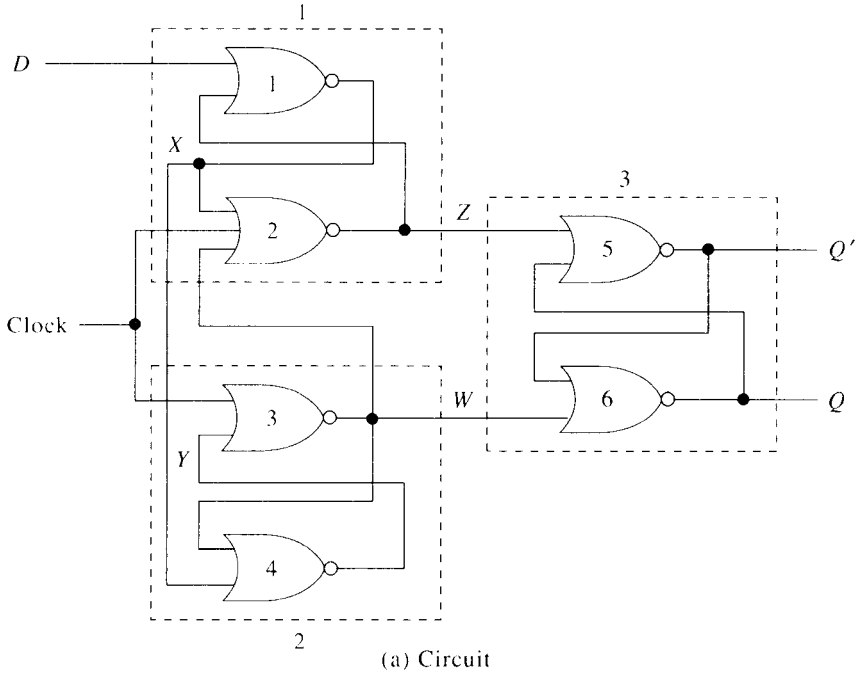



**Figure 2.17** Trailing edge-triggered flip-flop

at gate 1. When $W$ is 1, $D$ is blocked at gates 2 and 4. This blocking requires one gate delay after the trailing edge of the clock, and hence $D$ should not change until this blocking occurs. Thus, the *hold* time is one gate delay.

Note that the total time required for the flip-flop transition is three gate delays after the trailing edge – one gate delay for $W$ and $Z$ to change and two gate delays after that for $Q$ and $Q'$ to change. Thus, if we add $t_{\text{setup}}$ of two gate delays to the transition time of three gate delays, the minimum clock period is five gate delays if the output of the flip-flop is fed directly back to its input. If additional circuitry is in the feedback path, as is usually the case with most sequential circuits, the minimum clock period increases correspondingly.

A leading edge-triggered flip-flop can be designed using cross-coupled NAND circuits along the lines of the circuit shown above.

## 2.3 FLIP-FLOP ICs

Appendix C provides the details of several flip-flop ICs. TTL 7474 is a dual positive edge-triggered D flip-flop IC. The triangle at the clock input in the graphic symbol indicates positive edge triggering (Negative edge triggering is indicated by a triangle along with a bubble at the input as shown in the case of 74LS73. $S_D$ and $R_D$ are active-low asynchronous set and reset inputs, respectively, and operate independently of the clock. The date on the D input are transferred to the Q output at the positive clock edge. The D input must be stable one setup time (20 ns) prior to the positive edge of the clock. The positive transition time of the clock (i.e. from 0.8 to 2.0 V) should be equal to or less than the clock-to-output delay time for the reliable operation of the ff.

The 7473 and 74LS73 are dual master–slave JK flip-flop ICs. The 7473 is positive pulse-triggered (note the absence of the triangle on the clock input in the graphic symbol). JK information is loaded into the master while the clock is high and transferred to the slave during the clock high-to-low transition. For the conventional operation of this flip-flop, the JK inputs must be stable while the clock is high. The flip-flop also has direct set and reset inputs.

The 74LS73 is a negative edge-triggered flip-flop. The JK inputs must be stable one setup time (20 ns) prior to the high-to-low transition of the clock. This flip-flop has an active-low direct reset input.

The 7475 has four bistable latches. Each two-bit latch is controlled by an active-high enable input (E). When enabled, the data enter the latch and appear at the Q outputs. The Q outputs follow the data inputs as long as the enable is high. The latched outputs remain stable as long as the enable input

stays low. The data inputs must be stable one setup time prior to the high-to-low transition of the enable, for the data to be latched.

## 2.4 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUITS

The analysis of a synchronous sequential circuit is the process of determining the functional relation that exists between its outputs, its inputs, and its internal states. The contents of all the flip-flops in the circuit combined determine the internal state of the circuit. Thus, if the circuit contains $n$ flip-flops, it can be in one of the $2^n$ states. Knowing the present state of the circuit and the input values at any time $t$, we should be able to derive its next state (i.e., the state at time $t + 1$) and the output produced by the circuit at $t$.

A sequential circuit can be described completely by a state table that is very similar to the ones shown for flip-flops in Figs. 2.4 through 2.10. For a circuit with $n$ flip-flops, there will be $2^n$ rows in the state table. If there are $m$ inputs to the circuit, there will be $2^m$ columns in the state table. At the intersection of each row and column, the next-state and the output information are recorded. A *state diagram* is a graphical representation of the state table, in which each state is represented by a circle and the state transitions are represented by arrows between the circles. The input combination that brings about the transition and the corresponding output information are shown on the arrow. Analyzing a sequential circuit thus corresponds to generating the state table and the state diagram for the circuit. The state table or state diagram can be used to determine the output sequence generated by the circuit for a given input sequence if the *intial state* is known. It is important to note that for proper operation, a sequential circuit must be in its initial state before the inputs to it can be applied. Usually the power-up circuits are used to initialize the circuit to the appropriate state when the power is turned on. The following examples will illustrate the analysis procedure.

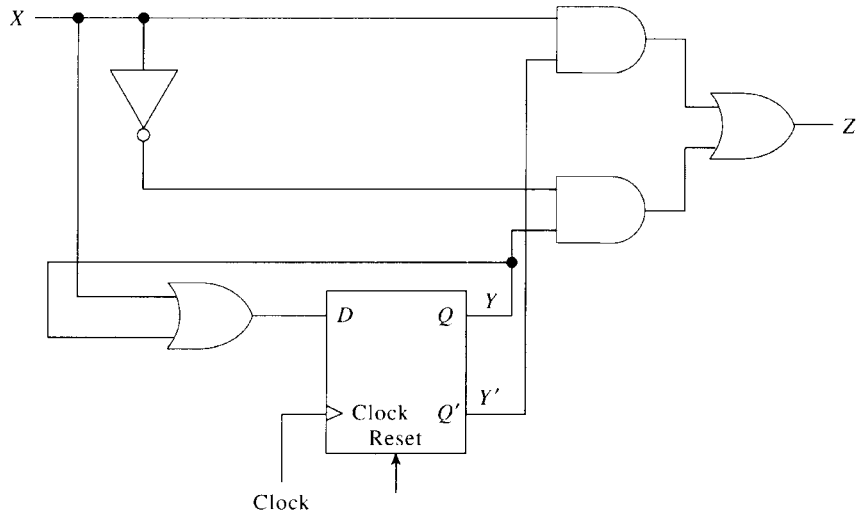
---

**Example 2.1** Consider the sequential circuit shown in Fig. 2.18(a). There is one circuit input $X$ and one output $Z$, and the circuit contains one $D$ flip-flop. To analyze the operation of this circuit, we can trace through the circuit for various input values and states of the flip-flop to derive the corresponding output and the next-state values. Since the circuit has one flip-flop, it has two states (corresponding to $Q = 0$ and $Q = 1$). The present state is designated as $Y$ (in this circuit, $Y = Q$). The output $Z$ is a function of the state of the circuit $Y$ and the input $X$ at any time $t$. The next state of the

circuit $Y(t+1)$ is determined by the value of the $D$ input at time $t$. Since the memory element is a $D$ flip-flop, $Y(t+1) = D(t)$.

Assume that $Y(t) = 0$ and $X(t) = 0$. Tracing through the circuit, we can see that $Z = 0$ and $D = 0$. Hence, $Z(t) = 0$ and $Y(t+1) = 0$. The above state transition and output are shown in the top left blocks of the next-state and output tables in (b). Similarly, when $X(t) = 0$ and $Q(t) = 1$, $Z(t)$ and $D(t) = 1$, making $Y(t+1) = 1$, as shown in the bottom left blocks of these tables. The other two entries in these tables are similarly derived by tracing through the circuit. The two tables are merged into one, entry by entry, as shown in (c) to form the so-called *transition table* for the circuit. Each block of this table corresponds to a present state and an input combination. Corresponding next-state and output information is entered in each block, separated by a slash mark. From the table in (c), we can see that if the state of the circuit is 0, it produces an output of 0 and stays in the state 0 as long as the input values are 0s. The first 1 input condition sends the circuit to a 1 state with an output of 1. Once it is in the 1 state, the circuit remains in that state regardless of what the input is, but the output is the complement of the input $X$. The state diagram in (d) illustrates the operation of the circuit graphically. Here, each circle represents a state, and an arrow represents the state transition. The input value corresponding to that transition and the output of the circuit at that time are represented on each arrow, separated by a slash. We will generalize the state diagram and state table representations in the next example.

Since the flip-flop is a positive edge-triggered flip-flop, the state transition takes place only when the rising edge of the clock occurs. However, this fact cannot be explicitly shown in the above tables. A timing diagram can be used to illustrate these timing characteristics. The operation of the circuit for a four-clock pulse period is shown in (e). The input $X$ is assumed to make the transitions shown. The flip-flop is positive edge-triggered. The state change occurs as a result of the rising edge but takes a certain amount of time after the edge occurs. It is assumed that the new state is attained by the falling edge of the clock. Assuming an initial state of 0, at $t_1$, $D$ is 0 and $Z$ is 0, thus not affecting $Y$. At $t_2$, $X$ changes to 1 and hence $D$ and $Z$ change to 1, neglecting the gate delays. At $t_3$, the positive edge of the clock starts the state transition, making $Q$ reach 1 by $t_4$. $Z$ goes to 0, since $Q$ goes to 1 at $t_4$. $X$ changes to 0 at $t_5$, thereby bringing $Z$ to 1 but not changing $D$. Hence $Y$ remains 1 through $t_8$, as does $D$. Corresponding transitions of $Z$ are also shown. The last part of the timing diagram shows $Z$ ANDed with the clock to illustrate the fact that output $Z$ is valid only during the clock pulse. If $X$ is assumed to be valid only during clock pulses, the timing diagram represents an input sequence of $X = 0101$ and the corresponding output sequence of $Z = 0110$. Note that the output sequence is the twos complement of the input sequence with the LSB occurring first and the MSB last.




(a) Circuit

*Notes*:

1. Arrows are shown on the rising edge of the clock to emphasize that the circuit uses a flop-flop triggered by that edge.
2. The clock at $t_3$ triggers $Q$ to change from 1 to 0, since $X$ is 1. But the state change is delayed by $\Delta t$ and occurs before the corresponding falling edge of the clock. For simplicity, we have assumed that $\Delta t =$ clock width. Hence the transition of $Q$ is shown at $t_4$. In practice, the clock width will be slightly larger than $\Delta t$.
3. We will also ignore the data setup and hold times in timing diagrams in this chapter for simplicity. We will simply use the value of the flip-flop input(s) at the triggering clock-edge to determine the transition of the flip-flop.

**Figure 2.18** Sequential circuit analysis

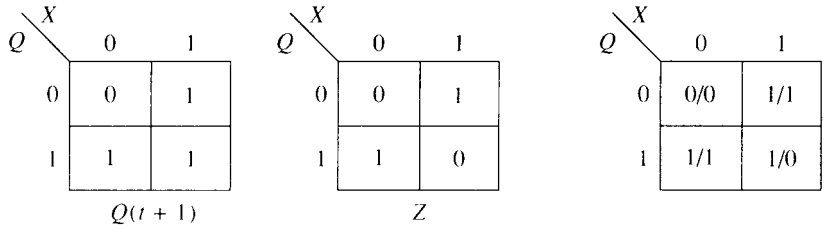
The circuit tracing procedure discussed here can be adopted for the analysis of simple circuits. As the circuit becomes more complex, the tracing becomes cumbersome. We will now illustrate a more systematic procedure for the derivation of the state table (and hence the state diagram).

From the analysis of the combinational circuit of Fig. 2.18(a), the flip-flop *input equation* (or *excitation equation*) is:

$$D = X + Y$$

and the circuit *output equation* is:

$$Z = XY' + X'Y$$



| $Q$ \ $X$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

$Q(t+1)$

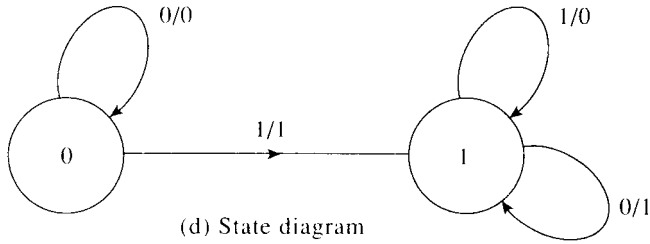
| $Q$ \ $X$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

$Z$

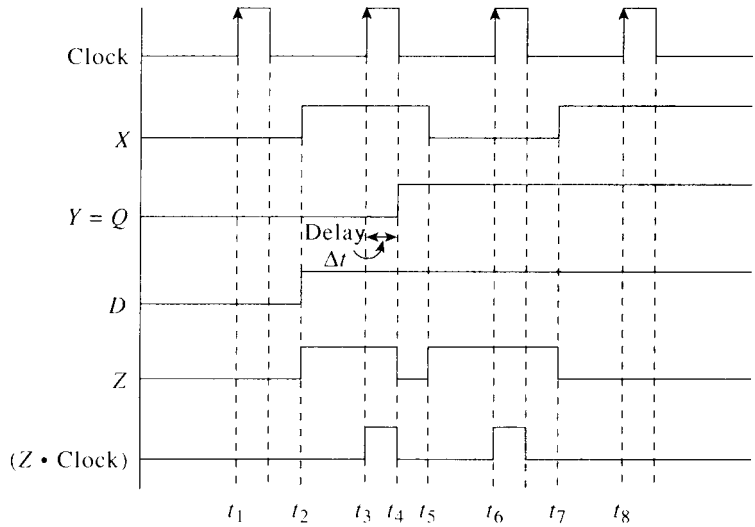
(b) Next-state and output tables

| $Q$ \ $X$ | 0 | 1 |
|---|---|---|
| 0 | 0/0 | 1/1 |
| 1 | 1/1 | 1/0 |

(c) Transition table




(d) State diagram




(e) Timing diagram for level input

**Figure 2.18** (Continued)

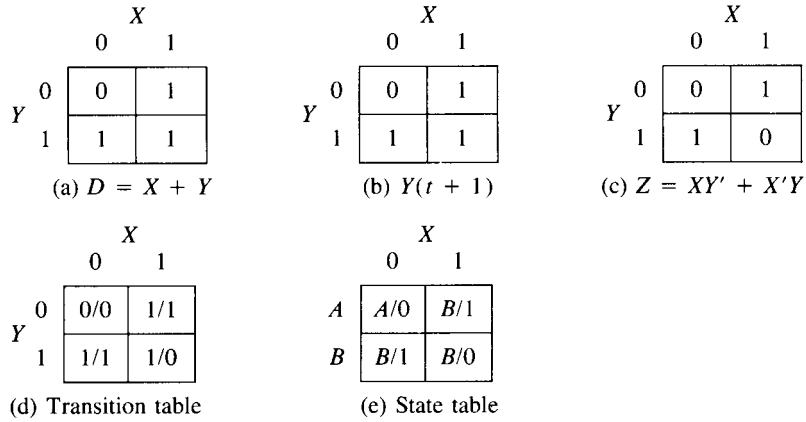



**Figure 2.19** Sequential circuit analysis

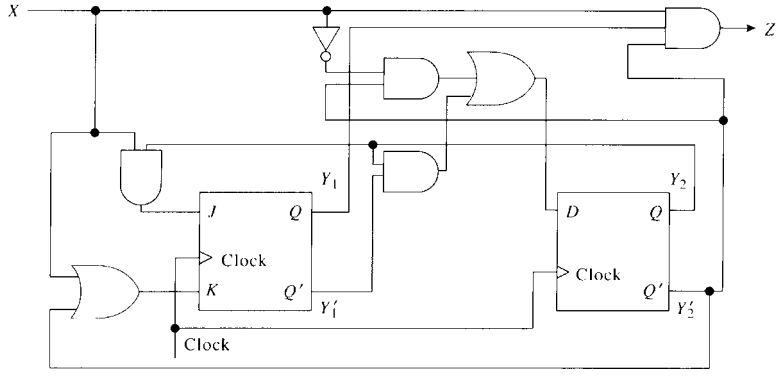
These equations express each input of the flip-flops in the circuit and the circuit output as functions of the circuit inputs and the state of the circuit at time $t$.

Figure 2.19(a) shows the truth table for $D$ in the form of a table whose rows correspond to the present state $Y$ of the circuit and whose columns correspond to the combination of values for the circuit input. Knowing the value of $D$, at each block in this table (i.e., at each present-state-input pair), we can determine the corresponding next state $Y(t+1)$ of the flip-flop, using the $D$ flip-flop excitation table. In the case of the $D$ flip-flop, the next state is the same as $D$; hence the next-state table shown in (b) is identical to (a). The truth table for the output $Z$ is represented by the table (c). Tables (b) and (c) are merged to form (d), which is usually called a *transition table*, since it shows the state and output transitions in the binary form. In general, each state in the circuit is designated by an alphabetic character. The transition table is converted into the *state table* by the assignment of alphabets to state transitions. The state table obtained by assigning $A$ to 0 and $B$ to 1 is shown in (e).

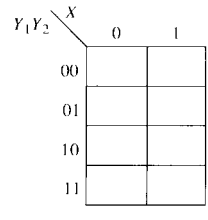
Example 2.2 illustrates the analysis procedure for a more complex circuit.

---

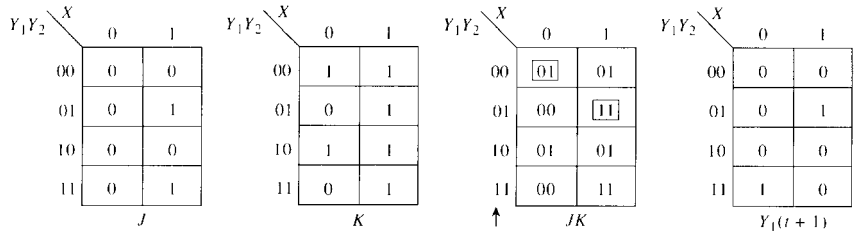
**Example 2.2** Consider the sequential circuit shown in Fig. 2.20(a). There are two clocked flip-flops, one input line $X$, and one output line $Z$. The $Q$ outputs of the flip-flops ($Y_1$, $Y_2$) constitute the present state of the circuit at




(a) Circuit diagram




(b) State table format

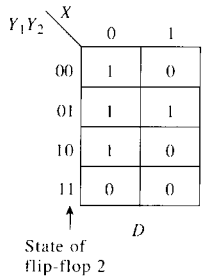


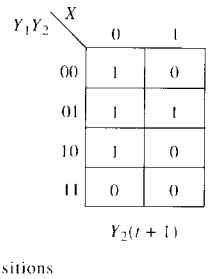

(c) *JK* flip-flop transitions

**Figure 2.20** A sequential circuit (*continued*)

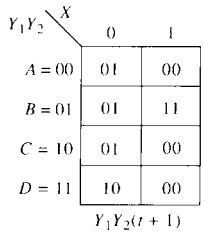
any time $t$. The signal values $J$ and $K$ determine the next state $Y_1(t+1)$ of the $JK$ flip-flop. The value of $D$ determines the next state $Y_2(t+1)$ of the $D$ flip-flop. Since both flip-flops are triggered by the same clock, their transitions occur simultaneously.

In practice, only one type of flip-flop is used in a circuit. Since an IC generally contains more than one flip-flop, using a single type of flip-flop in the circuit reduces the component count and hence the cost of the circuit. Different types of flip-flops have been used in the examples in this chapter for illustration purposes only.

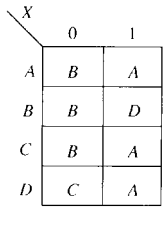


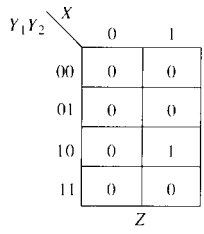

(d) $D$ flip-flop transitions



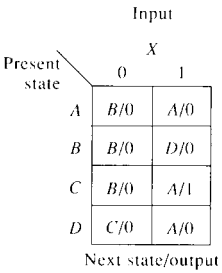

(c) Transition table



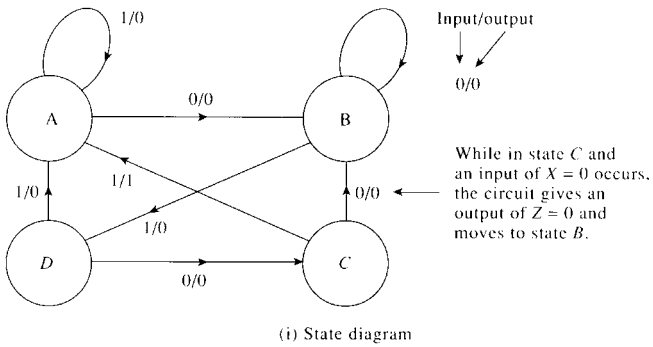

(f) Next state table



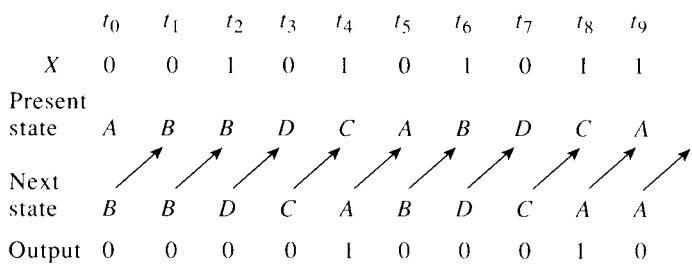

(g) Output




(h) State table




(i) State diagram




(j) An I/O sequence

**Figure 2.20** (Continued)

By analyzing the combinational portion of the circuit, we can derive the flip-flop input (or excitation) equations:

$$J = XY_2 \text{ and } K = X + Y_2'$$

and

$$D = Y_1' Y_2 + X' Y_2'$$

The circuit output equation is

$$Z = XY_1 Y_2'$$

Because there are two flip-flops, there will be four states, and hence the state table shown in (b) will have four rows. The rows are identified with the *state vectors* $Y_1 Y_2 = 00$, 01, 10, and 11. Input $X$ can be 0 or 1, and hence the state table will have two columns.

The next-state transitions of the *JK* flip-flop are derived in (c). Note again that the tables shown in (c) for $J$ and $K$ are simply the rearranged truth tables that reflect the combination of input values along the columns and the combination of present-state values along the rows. Tables for $J$ and $K$ are then merged, entry by entry, to derive the composite *JK* table – which makes it easier to derive the state transition of the *JK* flip-flop. Although both $Y_1(t)$ and $Y_2(t)$ values are shown in this table, only the $Y_1(t)$ value is required to determine $Y_1(t+1)$ once the $J$ and $K$ values are known. For example, in the boxed entry at the top left of the table, $J = 0$ and $K = 1$; hence, from the characteristic table for the *JK* flip-flop (Fig. 2.12), the flip-flop will reset, and $Y_1(t+1)$ will equal 0. Similarly, in the boxed entry in the second row, $J = 1$ and $K = 1$. Hence, the flip-flop complements its state. Since the $Y_1(t)$ value corresponding to this entry is 0, $Y_1(t+1) = 1$. This process is repeated six more times to complete the $Y_1(t+1)$ table.

The analysis of *D* flip-flop transitions is shown in (d); $Y_2(t+1)$ is derived from these transitions.

The transition tables for the individual flip-flops are then merged, column by column, to form the transition table in (e) for the entire circuit. Instead of denoting the states by binary state vectors, letter designations can be used for each state, as shown in (e), and the next-state table shown in (f) is derived. The output table in (g) is derived from the circuit output equation shown above. The output and next-state tables are then merged to form the state table (h) for the circuit. The state table thoroughly depicts the behavior of the sequential circuit. The state diagram for the circuit derived from the state table is shown in (i).

Assuming a starting (or initial) state of $A$, the input sequence and the corresponding next-state and output sequences are shown in (j). Note that

the output sequence indicates that the output is 1 only when the circuit input sequence is 0101. Thus, this is a 0101 sequence detector.

Note that once a sequence is detected, the circuit goes into the starting state $A$ and another complete 0101 sequence is required for the circuit to produce an output of 1. Can the state diagram in the above example be rearranged to make the circuit detect *overlapping* sequences? That is, the circuit should produce an output if a 01 occurs directly after the detection of a 0101 sequence. For example.

$$X = 000101010100101$$
$$Z = 000001010100001$$

The state diagram shown in Fig. 2.21 accomplishes this.

If the *starting* or the initial state of the circuit and the input sequence are known, the state table and the state diagram for a sequential circuit permit a functional analysis whereby the circuit's behavior can be determined. Timing analysis is required when a more detailed analysis of the circuit parameters is needed. Figure 2.22 shows the timing diagram for the first five clock pulses in Example 2.2.

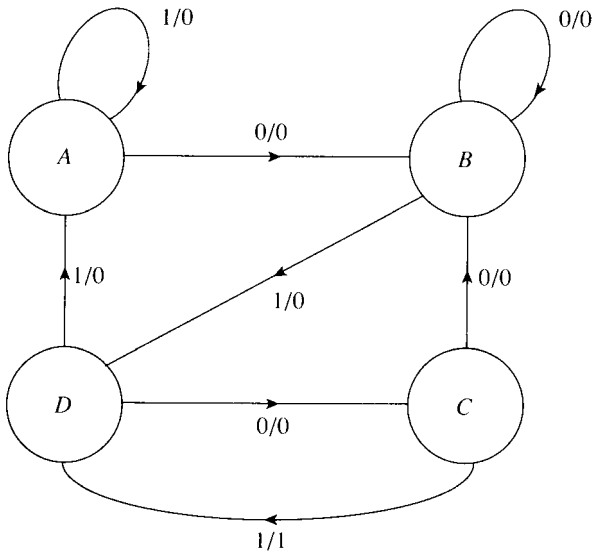



**Figure 2.21** A 0101 overlapped sequence detector

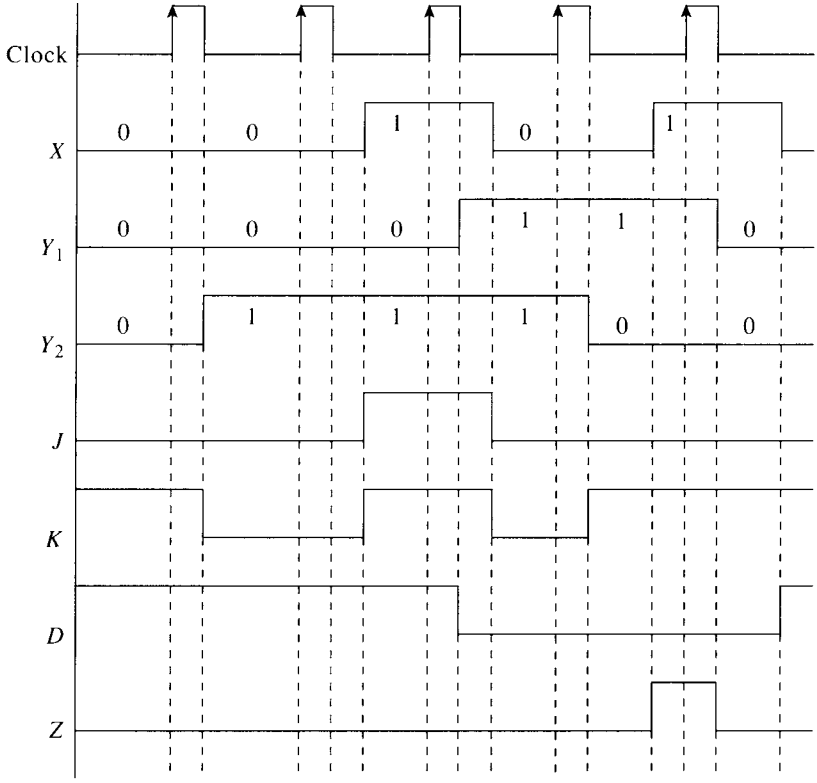



**Figure 2.22** Timing diagram for Example 2.2

This heuristic analysis can be formalized into the following step-by-step procedure for the analysis of synchronous sequential circuits:

1. Analyze the combinational part of the circuit to derive excitation equations for each flip-flop and the circuit output equations.
2. Note the number of flip-flops ($p$) and determine the number of states ($2^p$). Express each flip-flop input equation as a function of circuit inputs and the present state and derive the transition table for each flip-flop, using the characteristic table for the flip-flop.
3. Derive the next-state table for each flip-flop and merge them into one, thus forming the transition table for the entire circuit.
4. Assign names to state vectors in the transition table to derive the next-state table.

5. Using the output equations, draw a truth table for each output of the circuit, and rearrange these tables into the state table form. If there is more than one output, merge the output tables column by column to form the circuit output table.
6. Merge the next-state and output tables into one to form the state table for the entire circuit.
7. Draw the state diagram.

It is not always necessary to follow this analysis procedure. Some circuits yield a more direct analysis, as shown in Example 2.3.

---

**Example 2.3** Figure 2.23(a) shows a sequential circuit made up of two $T$ flip-flops. Recall that the $T$ flip-flop complements its state when the $T$ input is 1. Hence, if input $X$ is held at 1, $FF0$ complements at each clock pulse, while $FF1$ complements only when $Q_0$ is 1 (i.e., every second clock pulse). The state diagram is shown in (b). Note that the output of the circuit is the state itself. As can be seen in (b), this is a modulo-4 counter. Refer to Example 2.5 for another modulo-4 counter design.

(What would be the count sequence if $FF0$ and $FF1$ in Fig. 2.23 were falling-edge triggered?)

---

## 2.5 DESIGN OF SYNCHRONOUS SEQUENTIAL CIRCUITS

The design of a sequential circuit is the process of deriving a logic diagram from the specification of the circuit's required behavior. The circuit's beha-

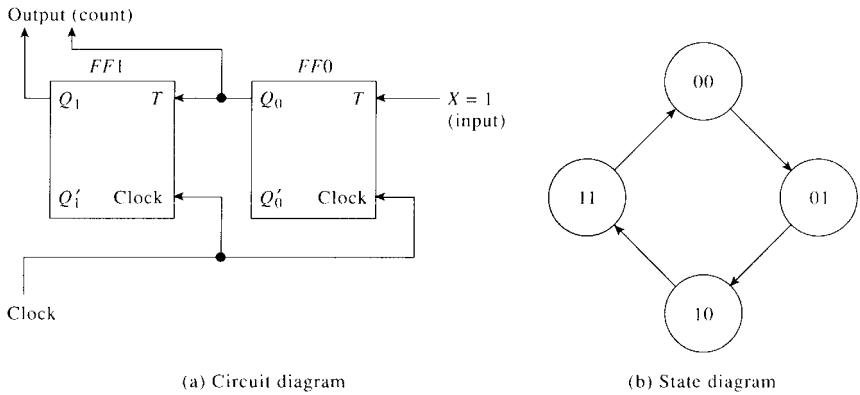



**Figure 2.23** A modulo-4 counter

vior is often expressed in words. The first step in the design is then to derive an exact specification of the required behavior in terms of either a state diagram or a state table. This is probably the most difficult step in the design, since no definite rules can be established to derive the state diagram or a state table. The designer's intuition and experience are the only guides. Once the description is converted into the state diagram or a state table, the remaining steps become mechanical. We will examine the classical design procedure through the examples in this section. It is not always necessary to follow this classical procedure, as some designs lend themselves to more direct and intuitive design methods.

The classical design procedure consists of the following steps:

1. Deriving the state diagram (and state table) for the circuit from the problem statement.
2. Deriving the number of flip-flops ($p$) needed for the design from the number of states in the state diagram, by the formula

   $$2^{p-1} < n \leq 2^p$$

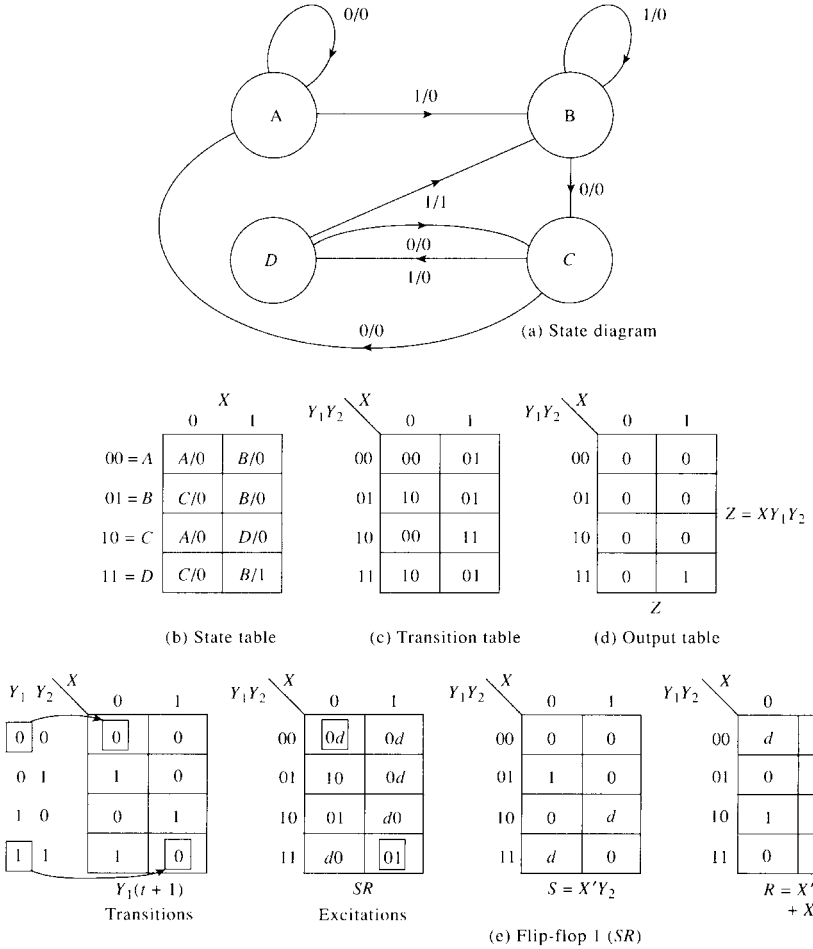
   where $n$ = number of states.
3. Deciding on the types of flip-flops to be used. (This often simply depends on the type of flip-flops available for the particular design.)
4. Assigning a unique $p$-bit pattern (state vector) to each state.
5. Deriving the state transition table and the output table.
6. Separating the state transition table into $p$ tables, one for each flip-flop.
7. Deriving an input table for each flip-flop input using the excitation tables (Fig. 2.13).
8. Deriving input equations for each flip-flop input and the circuit output equations.
9. Drawing the circuit diagram.

This design procedure is illustrated by the following examples.

---

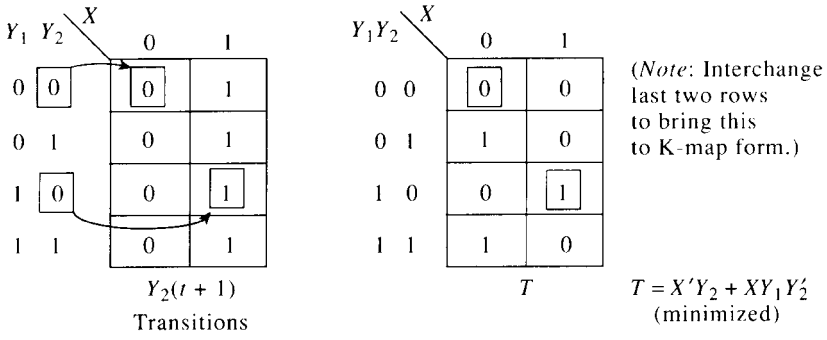
**Example 2.4** Design a sequential circuit that detects an input sequence of 1011. The sequences may overlap. A 1011 sequence detector gives an output of 1 when the input completes a sequence of 1011. Because overlap is allowed, the last 1 in the 1011 sequence could be the first bit of the next 1011 sequence, and hence a further input of 011 is enough to produce an output of 1. That is, the input sequence 1011011 consists of two overlapping sequences.

Figure 2.24(a) shows a state diagram. The sequence starts with a 1. Assuming a starting state of $A$, the circuit stays in $A$ as long as the input is 0, producing an output of 0 waiting for an input of 1 to occur. The first 1 input takes the circuit to a new state $B$. So long as the inputs continue to be 1, the circuit has to stay in $B$ waiting for a 0 to occur to continue the sequence and hence to move to a new state $C$. While in $C$, if a 0 is received, the sequence of

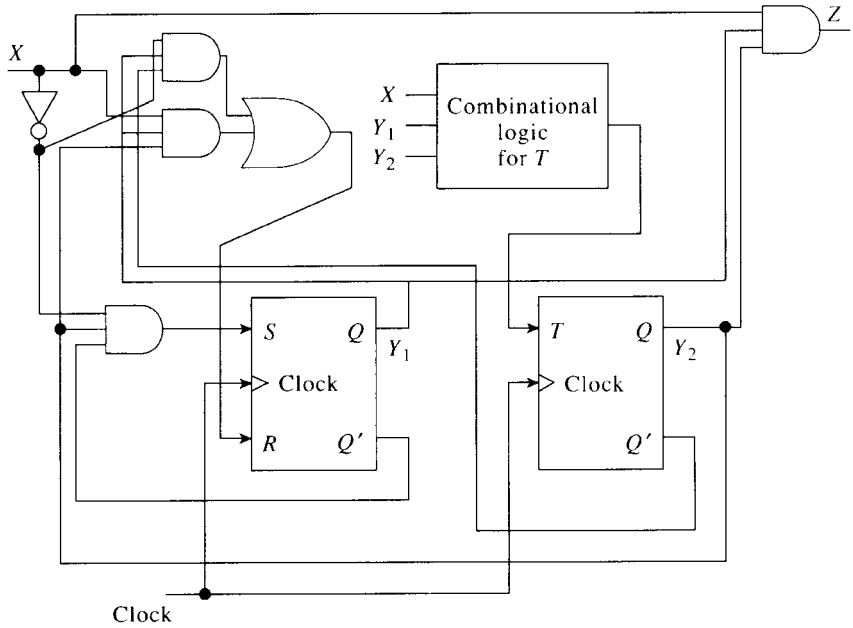



*Note*: Interchange the last two rows of $S$ and $R$ tables to bring them in to the form of K-map and then minimize.

**Figure 2.24** 1011 sequence detector




(f) Flip-flop 2 ($T$)




(g) Circuit diagram

**Figure 2.24** (Continued)

inputs is 100, and the current sequence cannot possibly lead to 1011. Hence, the circuit returns to state $A$. But if a 1 is received while in $C$, the circuit moves to a new state $D$, continuing the sequence. While in $D$, a 1 input completes the 1011 sequence. The circuit gives a 1 output and goes to $B$ in preparation for a 011 for a new sequence. A 0 input while at $B$ creates the possibility of an overlap, and hence the circuit returns to $C$ so that it can detect the 11 subsequence required to complete the sequence.

Drawing a state diagram is purely a process of trial and error. In general, we start with an initial state. At each state, we move either to a new state or to one of the already-reached states, depending on the input values. The state diagram is complete when all the input combinations are tested and accounted for at each state. Note that the number of states in the diagram cannot be predetermined and various diagrams typically are possible for a given problem statement. The amount of hardware needed to synthesize the circuit increases with the number of states. Thus, it is desirable to reduce the number of states if possible.

The state table for the example is shown in Fig. 2.24(b). Since there are four states, we need two flip-flops. The four two-bit patterns are arbitrarily assigned to the states, and the transition table in (c) and the output table in (d) are drawn. From the output table, we can see that

$$Z = XY_1Y_2$$

We will use an $SR$ flip-flop and a $T$ flip-flop. It is common practice to use one kind of flip-flop in a circuit. Different kinds are used here for illustration purposes only. The transitions of flip-flop 1 ($SR$) extracted from (c) are shown in the first table $Y_1(t+1)$ of (e). From these transitions and using the excitation tables for the $SR$ flip-flop (Fig. 2.13), the $S$ and $R$ excitations are derived. (For example, the 0-to-0 transition of the flip-flop requires that $S = 0$ and $R = 3$, and a 1-to-0 transition requires that $S = 0$ and $R = 1$.) The $S$ and $R$ excitations (which are functions of $X$, $Y_1$, and $Y_2$) are separated into individual tables, and the excitation equations are derived. These equations are shown in (e). The input equation for the second flip-flop ($T$) is similarly derived and is shown in (f). The circuit diagram is shown in (g).

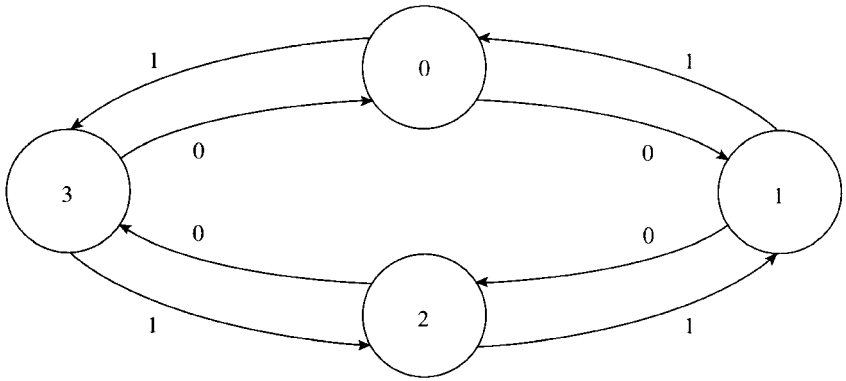
---

The complexity of a sequential circuit can be reduced by simplifying the input and output equations. In addition, a judicious allocation of state vectors to the states also reduces circuit complexity.

---

**Example 2.5** *Modulo-4 up-down counter*. The modulo-4 counter will have four states: 0, 1, 2 and 3. The input $X$ to the counter controls the direction of

the count: up if $X = 0$ and down if $X = 1$. The state of the circuit (i.e., the count) itself is the circuit output. The state diagram is shown here:




Derivation of a state diagram for this counter is straightforward, since the number of states and the transitions are completely defined by the problem statement. Note that only input values are shown on the arcs – the output of the circuit is the state of the circuit itself. We will need two flip-flops and the assignment of 2-bit vectors for states is also defined to be 00, 01, 10, and 11 to correspond to 0, 1, 2, and 3, respectively. The state table and transition table are shown below: the control signal is $X$; $X = 0$ indicates "count up"; $X = 1$ indicates "count down."

| Present state | $X=0$ | $X=1$ |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 2 | 0 |
| 2 | 3 | 1 |
| 3 | 0 | 2 |

| $Y_1Y_2$ | $X=0$ | $X=1$ |
|---|---|---|
| 00 | 01 | 11 |
| 01 | 10 | 00 |
| 10 | 11 | 01 |
| 11 | 00 | 10 |

We will use a *JK* flip-flop and a *D* flip-flop. The input equations are derived below, and Fig. 2.25 shows the circuit.

## 2.6 REGISTERS

A register is a storage device capable of holding binary data; it is a collection of flip-flops. An $n$-bit register is built of $n$ flip-flops. Figure 2.26 shows a 4-bit register built out of four *D* flip-flops. There are four input lines, $IN_1$, $IN_2$,

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 10 | 1 | 0 |
| 11 | 0 | 1 |

Transitions of flip-flop 1 ($JK$)

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | $0d$ | $1d$ |
| 01 | $1d$ | $0d$ |
| 10 | $d0$ | $d1$ |
| 11 | $d1$ | $d0$ |

Inputs $JK$

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 10 | $d$ | $d$ |
| 11 | $d$ | $d$ |

$J$
$J = X'Y_2 + XY_2'$

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | $d$ | $d$ |
| 01 | $d$ | $d$ |
| 10 | 0 | 1 |
| 11 | 1 | 0 |

$K$
$K = X'Y_2 + XY_2'$

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | 1 | 1 |
| 01 | 0 | 0 |
| 10 | 1 | 1 |
| 11 | 0 | 0 |

Transitions of flip-flop 2 ($D$)

| $Y_1Y_2$ \ $X$ | 0 | 1 |
|---|---|---|
| 00 | 1 | 1 |
| 01 | 0 | 0 |
| 10 | 1 | 1 |
| 11 | 0 | 0 |

Inputs $D$
$D = Y_2$ ($D$ is simplified by observation.)

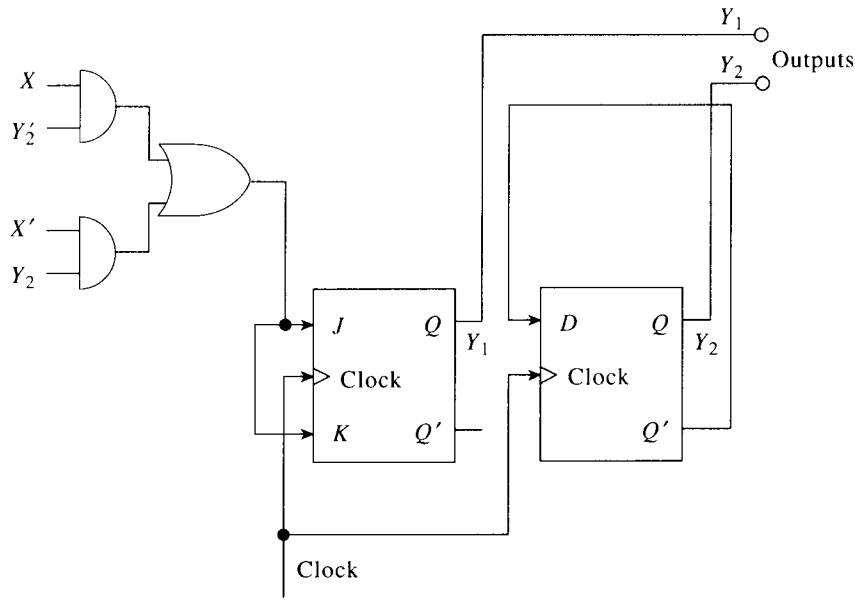



**Figure 2.25** Circuit for modulo-4 counter

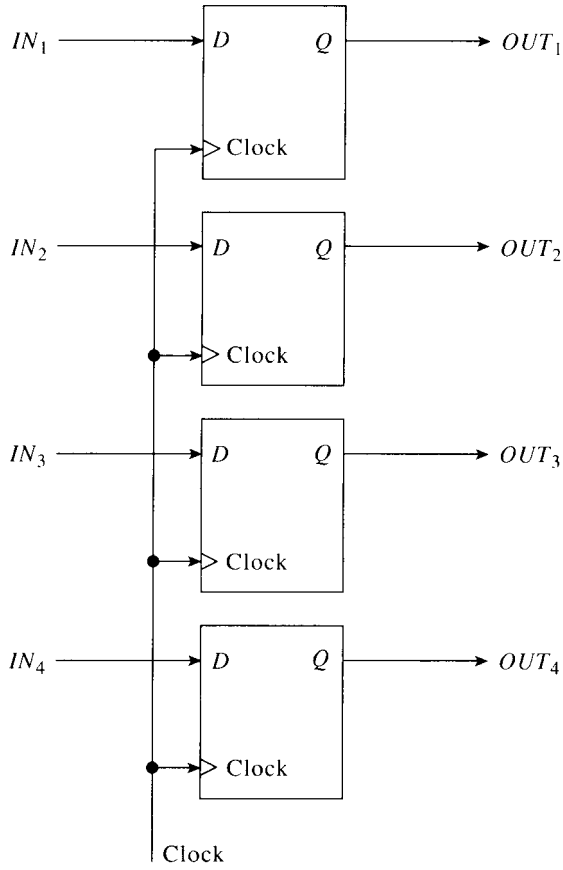



**Figure 2.26** A 4-bit register

$IN_3$, and $IN_4$, each connected to the $D$ input of the corresponding flip-flop. When a clock pulse occurs, the data from input lines $IN_1$ through $IN_4$ enter the register. The clock thus *loads* the register. The loading is in *parallel*, since all four bits enter the register simultaneously. $Q$ outputs of flip-flops are connected to output lines $OUT_1$ through $OUT_4$ and hence all four bits of data (i.e., contents of the register) are available simultaneously (i.e., in parallel) on the output lines. Hence, this is a parallel-input (parallel-load), parallel-output register.

At each clock pulse, a 4-bit data input enters the register from input lines $IN_1$ through $IN_4$ and remains in the register until the next clock pulse. The clock controls the loading of the register as shown in Fig. 2.27. LOAD must be 1 for data to enter the register. The CLEAR signal shown in Fig.

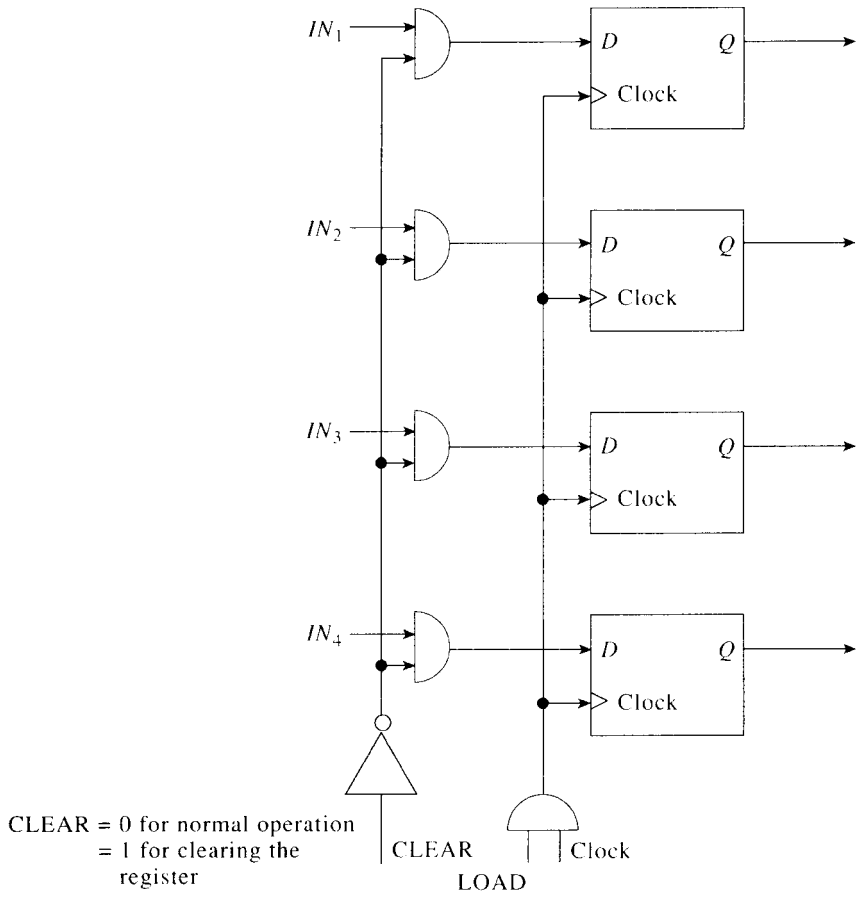
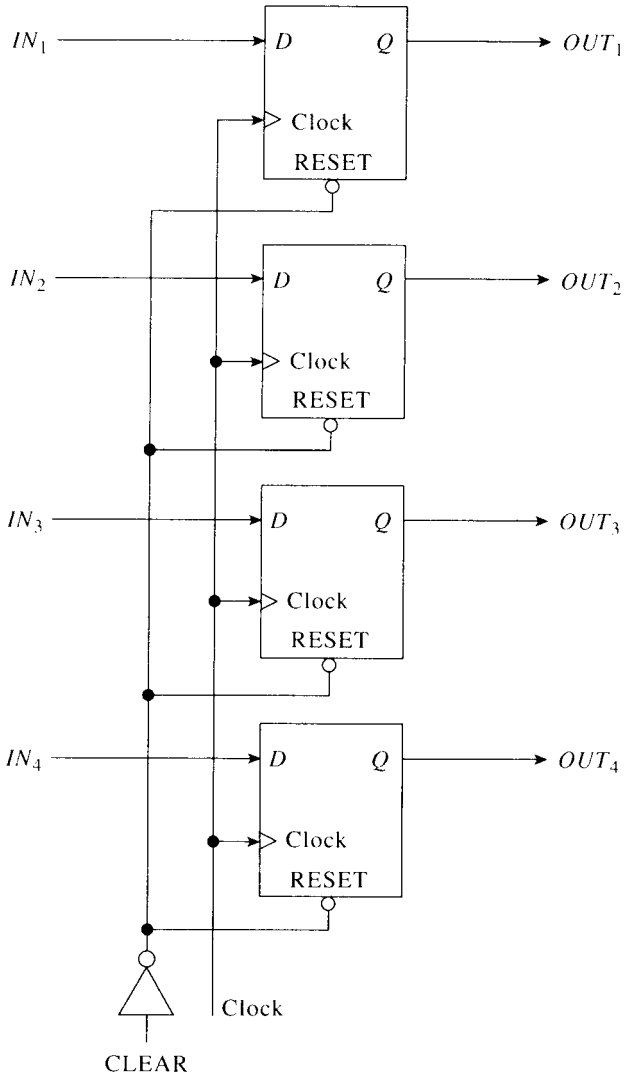



**Figure 2.27** A 4-bit register with CLEAR and LOAD

2.27 leads zeros into the register (i.e., *clears* the register). Clearing a register is a common operation and is normally done through the asynchronous clear input (RESET) provided on flip-flops. Thus, when asynchronous inputs are used, a clearing operation can be done independent of the clock. The CLEAR signal shown in Fig. 2.28 clears the register asynchronously. In this scheme, the CLEAR input must be set to 1 for clearing the register and should be brought to 0 to deactivate RESET and allow resumption of normal operation.

In the circuit of Fig. 2.28, the data on the input lines enter the register at each rising edge of the clock. Therefore, we need to make sure that the




Note: Low-active reset

**Figure 2.28** A 4-bit register with asynchronous CLEAR

data on input lines are always valid. Figure 2.29 shows a 4-bit register built out of *JK* flip-flops in which the data on input lines enters the register at the rising edge of the clock only when the LOAD is 1. When LOAD is 0, since both *J* and *K* will be 0, the contents of the register remain unchanged.

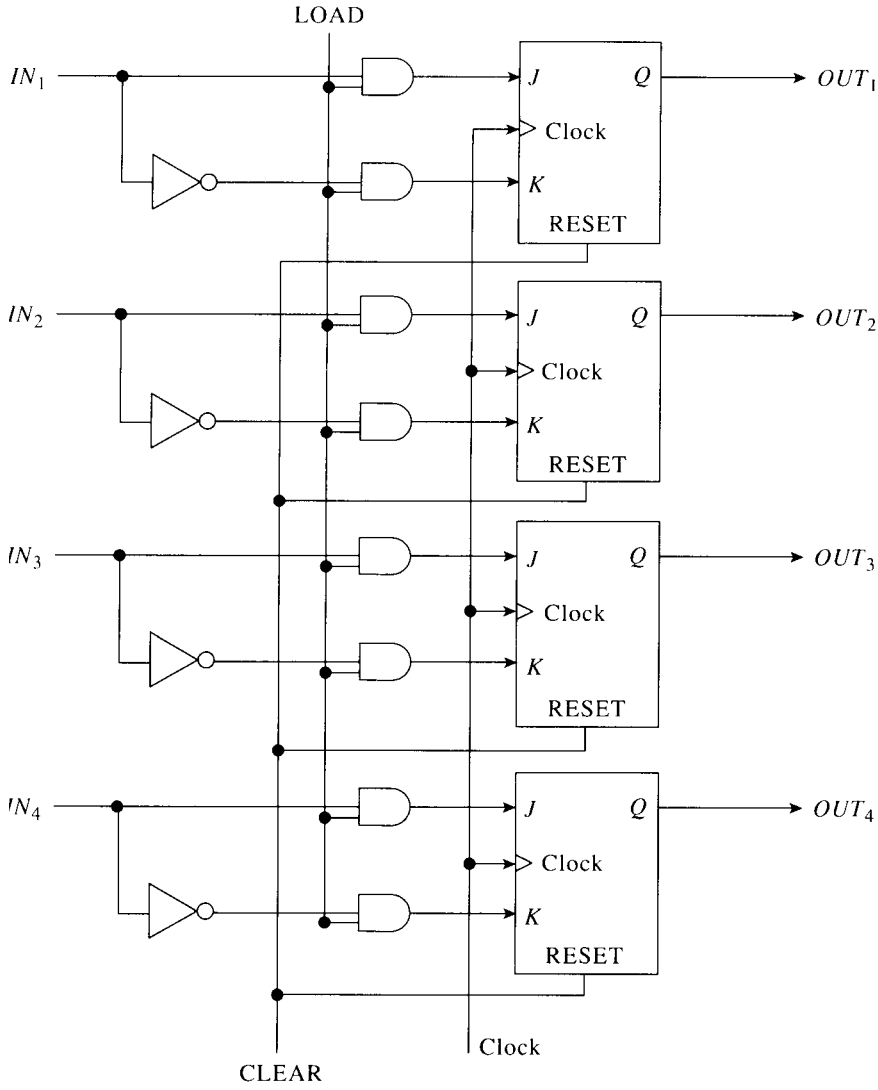



**Figure 2.29** A 4-bit register using *JK* flip-flops

Note that we have used two AND gates to gate the data in each flip-flop in Fig. 2.29. If we try to eliminate one of these gates by gating the IN line rather than $J$ and $K$ lines, we will be introducing unequal delays on $J$ and $K$ lines due to the extra NOT gate on the $K$ line. As long as the clock edge appears after both the inputs are settled, this unequal delay would not present a problem. If it does not appear, the circuit will not operate properly.

(Can you replace the $JK$ flip-flops in Fig. 2.29 with $D$ flip-flops? What changes are needed to retain the contents of the register unaltered when LOAD is 0, in this case?)

Choice of the type of flip-flop used in the circuit depends on the mode of implementation of the synchronous circuit. In designing integrated circuits, it is often more efficient to use $JK$ flip-flops with gated inputs. In implementing the circuit with MSI parts, it is common to use $D$ flip-flops with gated clocks.

A common operation on the data in a register is to *shift* it either right or left. Figure 2.30 shows a 4-bit *shift register* built out of $D$ flip-flops. The $Q$ output of each flip-flop is connected to the $D$ input of the flip-flop to its right. At each clock pulse, content of $d_1$ moves to $d_2$, content of $d_2$ moves to $d_3$, and that of $d_3$ moves into $d_4$, simultaneously. Hence, this is a *right*-shift register. The *output of the shift register at any time is the content of* $d_4$. If the *input* is set to 1, a 1 is entered into $d_1$ at each shift pulse. Similarly a 0 can be *loaded* by setting the input to 0.

An $n$-bit shift register can be loaded serially in $n$ clock pulses, and the contents of the register can be output serially using the *output* line in $n$ clock pulses. Note that in loading an $n$-bit right-shift register serially, the least significant bit must be entered first, followed by more significant bit values. Also, if the *output* line of the shift register is connected to its *input* line, the contents of the register "circulate" at each shift pulse.

Figure 2.31 shows a shift register with a serial-input, serial-output, parallel-output, circulate (left or right), and shift (left or right) capabilities.

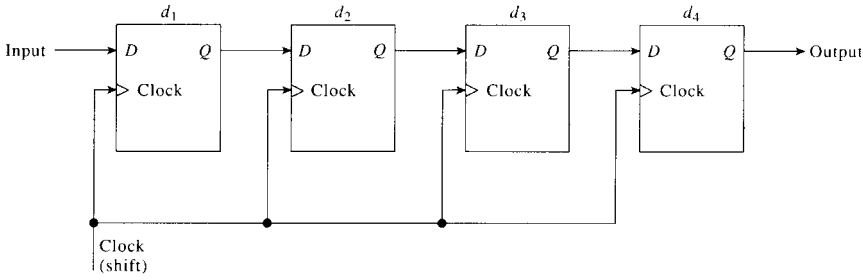



**Figure 2.30** A 4-bit shift register

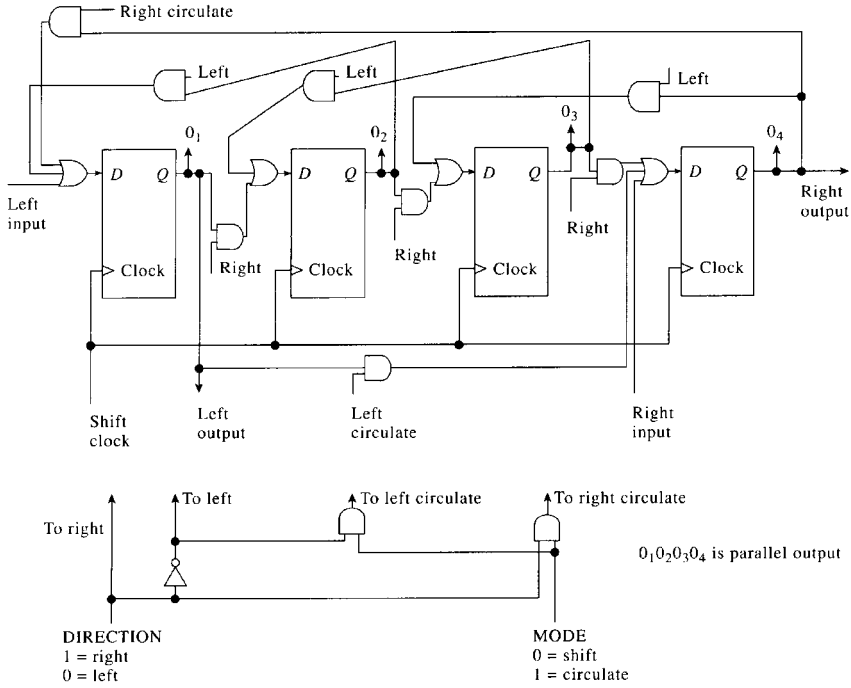



**Figure 2.31** A 4-bit universal shift register

Each *D* input can receive data from the flip-flop to the right or left of it, depending on whether the DIRECTION signal is 0 or 1, respectively. Since *right* and *left* signals are complements, the register can shift only in one direction at any time. The register performs shift or circulate based on the value of the MODE signal. When in shift mode, the data on left input enter the register if DIRECTION is 1, and the data on right input enter the register if DIRECTION is 0. The content of the register can be output in parallel through $0_1 0_2 0_3 0_4$ or in a serial mode through $0_4$.

A 3-bit shift register using *SR* flip-flops with right-shift and parallel or serial input capabilities is shown in Fig. 2.32. Refer to Appendix C for details of shift register ICs. The following examples illustrate the utility of shift registers in sequential circuit design.

---

**Example 2.6** *1011 sequence detector using a shift register*. It is possible to design sequential circuits without following the classical design procedure discussed earlier in this chapter. We will illustrate this by designing the 1011

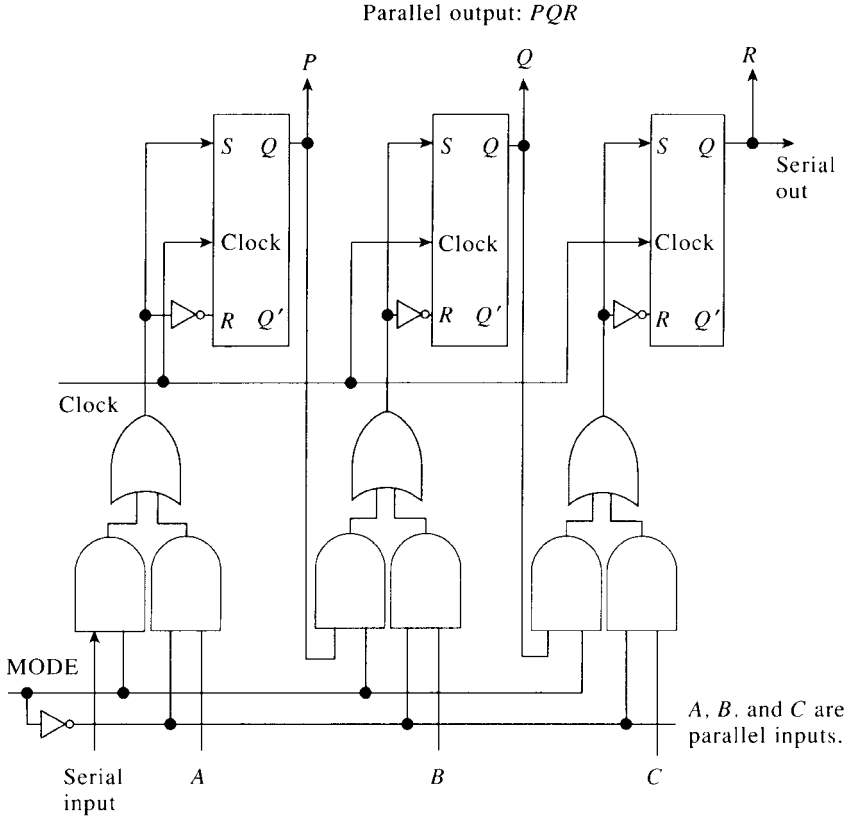



**Figure 2.32** A 3-bit shift register

sequence detector of Example 2.4, using a 4-bit right-shift register. The circuit is shown in Fig. 2.33(a). The input is $X$. The output $Z_1$ is when the shift register contains the sequence 1011 (i.e., 1101 left to right, since the input enters the shift register from the left). $Z_1$ is gated by the clock to produce $Z$. The same clock is used as the shift control for the shift register. Note that the shift register is activated by the rising edge of the clock.

The operation of the circuit is illustrated by the timing diagram in Fig. 2.33(b). At $t_1$, $X$ is 1. Hence a 1 enters the shift register. We will assume that the shift register contents are settled by $t_2$. Similarly, at $t_3$, a 0 enters the shift register, and at $t_5$ and $t_7$, 1s enter, resulting in the sequence 1011 being the

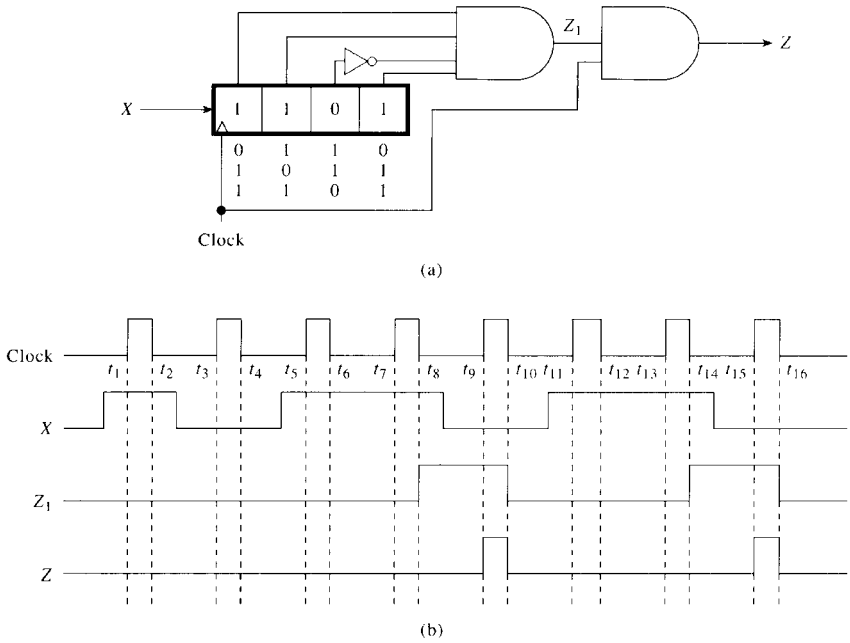



**Figure 2.33** 1011 sequence detector using a shift register (Example 2.6)

content of the shift register. Hence, $Z_1$ goes to 1 by $t_8$. Since a 0 enters the shift register at $t_9$, $Z_1$ will be 0 by $t_{10}$. Thus, $Z$ is 1 during $t_9$ and $t_{10}$. Note that $Z$ will be 1 again during $t_{15}$ and $t_{16}$, since the entry of 011 completes the sequence.

It is required that the shift register be cleared to begin with. Also note that this circuit requires four flip-flops compared to two flip-flops needed by the circuit in Fig. 2.24, while the combinational portion of this circuit is less complex.

---

**Example 2.7** *Serial adder*. The adder circuit designed in Section 1.6 uses $(n-1)$ full adders and one half-adder to generate the SUM of two $n$-bit numbers. The addition is done in parallel, although the CARRY has to propagate from the LSB position to the MSB position. This CARRY propagation delay determines the speed of the adder. If a slower speed of addition can be tolerated by the system, a *serial adder* can be utilized. The serial adder uses one full adder and two shift registers. The bits to be added

are brought to full adder inputs and the SUM output of the full adder is shifted into one of the operand registers while the CARRY output is stored in a flip-flop and is used in the addition of next most significant bits. The $n$-bit addition is thus performed in $n$ cycles (i.e., $n$ clock pulse times) through the full adder.

Figure 2.34 shows the serial adder for 6-bit operands stored in shift registers $A$ and $B$. The addition follows the stage-by-stage addition process (as done on paper) from LSB to MSB. The CARRY flip-flop is reset at the beginning of addition since the carry into the LSB position is 0. The full adder adds the LSBs of $A$ and $B$ with $C_{in}$ and generates SUM and $C_{out}$. During the first shift pulse, $C_{out}$ enters the CARRY flip-flop, SUM enters the MSB of $A$, and $A$ and $B$ registers are shifted right, simultaneously. Now the circuit is ready for the addition of the next stage. Six pulses are needed to complete the addition, at the end of which the least significant $n$ bits of the SUM of $A$ and $B$ will be in $A$, and the $(n+1)$th bit will be in the CARRY flip-flop. Operands A and B are lost at the end of the addition process.

If the LSB output of $B$ is connected to its MSB input, then $B$ will become a circulating shift register. The contents of $B$ are unaltered due to addition, since the bit pattern in $B$ after the sixth shift pulse will be the same as that before addition began. If the value of $A$ is also required to be

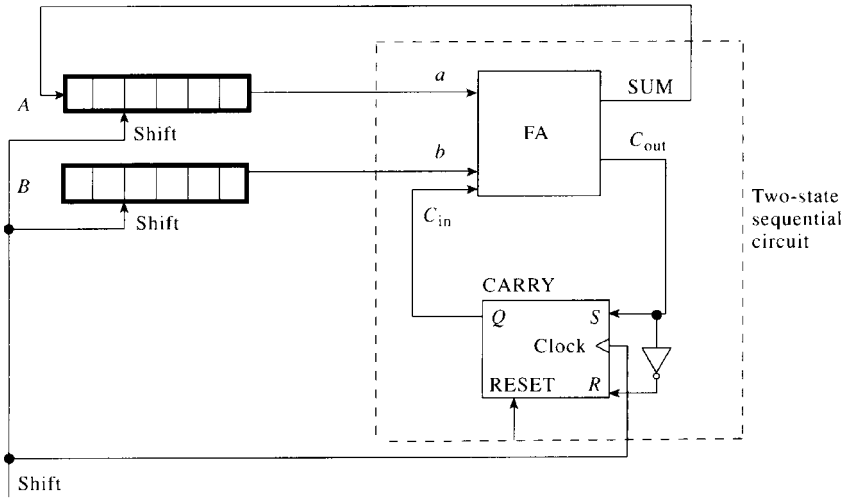



**Figure 2.34** Serial adder

preserved, $A$ should be converted into a circulating shift register and the SUM output of the full adder must be fed into a third shift register.

The circuit enclosed by dotted lines in Fig. 2.34 is a sequential circuit with one-flip-flop and hence two states, two input lines ($a$ and $b$) and one output line (SUM). $C_{in}$ is the present-state vector and $C_{out}$ is the next-state vector.

---

**Example 2.8** *Serial 2s complementer*. A serial 2s complementer follows the COPY-COMPLEMENT algorithm for 2s complementing the contents of a register (see Appendix A). The algorithm examines the bits of the register starting from the LSB. All consecutive zero bits as well as the first nonzero bit are "copied" as they are and the remaining bits until and including MSB are "complemented," to convert a number into its 2 complement. An example is given below.

| | | | |
|---|---|---|---|
| 1 0 1 1 0 1 0 | | 1 0 0 0 | An 11-bit number |
| COMPLEMENT | | COPY | |
| 0 1 0 0 1 0 1 | | 1 0 0 0 | Its 2s complement |

There are two distinct operations in this algorithm: COPY and COMPLEMENT. Further, the transition from a copying mode to complementing mode is brought about by the first nonzero bit. The serial complementer circuit must be a sequential circuit because the mode of operation at any time depends on whether the nonzero bit has occurred or not. There will be two states and hence one flip-flop in the circuit; one input line on which bits of the number to be complemented are entering starting with LSB; and one output line that is either the copy or the complement of the input. The circuit starts in the COPY state and changes to COMPLEMENT state when the first nonzero bit enters through the input line. At the beginning of each 2s complement operation, the circuit must be set to the COPY state.

Figure 2.35 shows the design of this circuit. From the state diagram in (a), the state table in (b) is derived, followed by the output equation in (c) and the input equations for the $SR$ flip-flop in (e). The circuit is shown in (f). The circuit is set to the COPY state by resetting the flip-flop before each complementation.

---

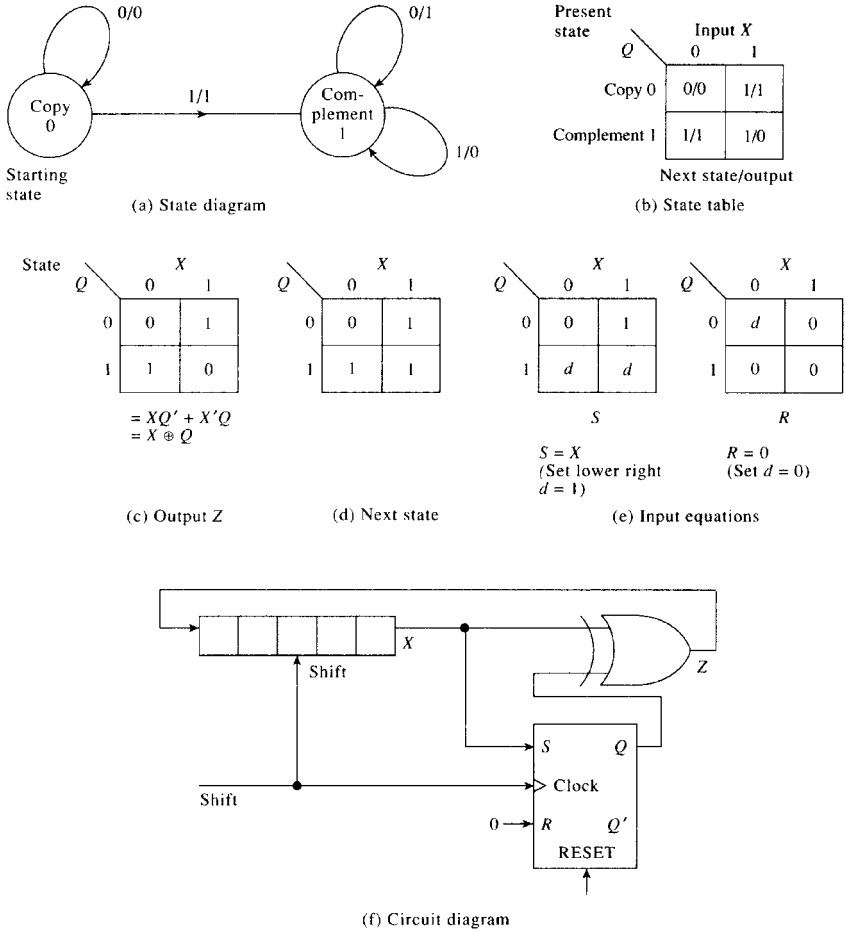



**Figure 2.35** Serial 2s complementer

## 2.7 REGISTER TRANSFER LOGIC

Manipulation of data in most digital systems involves the movement of data between registers. The data movement can be accomplished either in *serial* or in *parallel*. Transfer of an $n$-bit data from one register to the other (each of $n$ bits) takes $n$ shift pulses if done in serial mode, while it is done in one pulse time in parallel mode. A data path that can transfer 1 bit between the registers is sufficient for serial mode operation. This path is repeatedly used for transforming all $n$ bits one at a time. For a parallel, transfer scheme, $n$

such data paths are needed. Thus, a serial transfer scheme is less expensive in terms of hardware and slower than the parallel scheme.

Figure 2.36 shows the *parallel* transfer scheme from a 4-bit register $A$ to a 4-bit register $B$. Here, $X$ is a control signal. The data transfer occurs at the rising edge of the clock pulse only when $X$ is 1. When $X$ is 0, the $J$ and $K$ inputs of all the four flip-flops in register $B$ are at 0, and hence the contents of register $B$ remain unchanged even if the rising edge of the clock pulse occurs. In a synchronous digital circuit, control signals such as $X$ are also synchronized with the clock. The timing requirements for the proper operation of the parallel transfer circuit are shown in (b). At $t_1$, the control signal $X$ goes to 1, and at $t_2$ the register transfer occurs. $X$ can be brought to 0 at $t_2$.

We will represent the above transfer scheme by the diagram in (c). Each register is represented by a rectangle along with the clock input. The

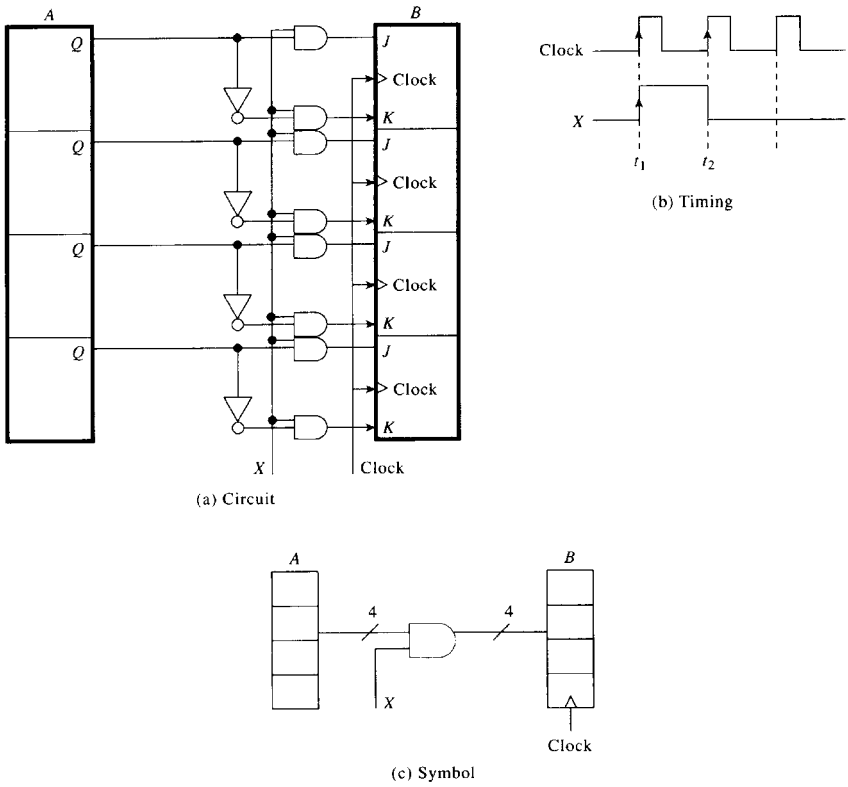
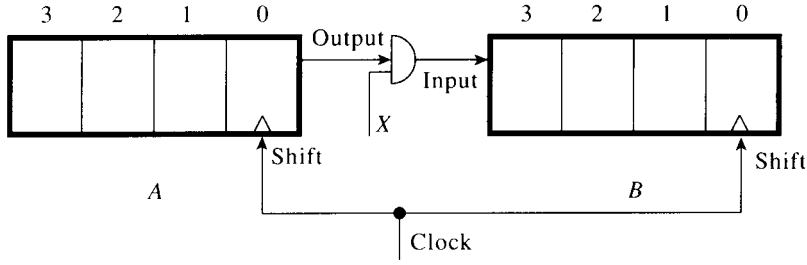


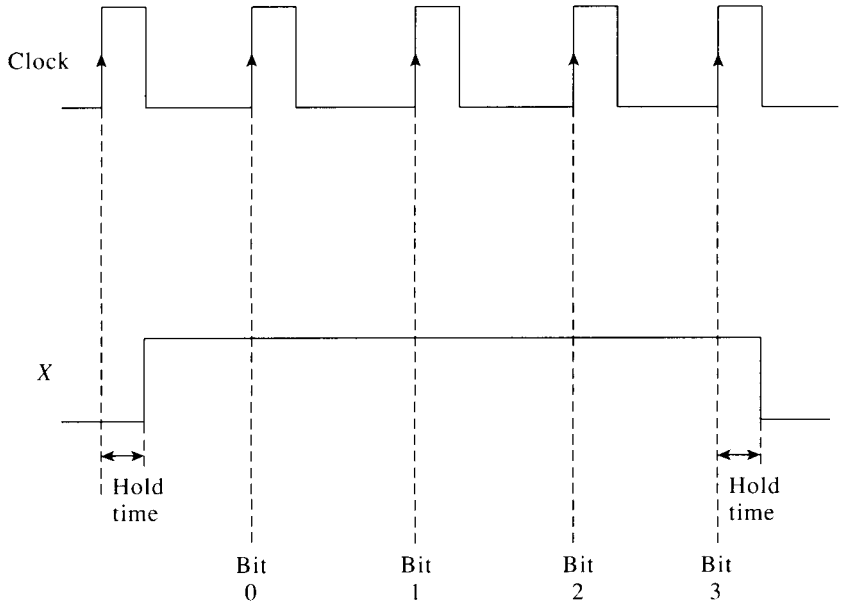

**Figure 2.36** Parallel transfer scheme

inputs and the outputs of the registers are shown as required. The number 4 shown next to the / indicates the number of bits transferred and hence the number of parallel lines needed, each line controlled by $X$. This is a common convention used to represent multiple bits of any signal in a circuit diagram.

Figure 2.37 shows the *serial* transfer scheme. Here, $A$ and $B$ are two 4-bit shift registers. They shift right in response to the shift clock. As seen by




(a) Circuit

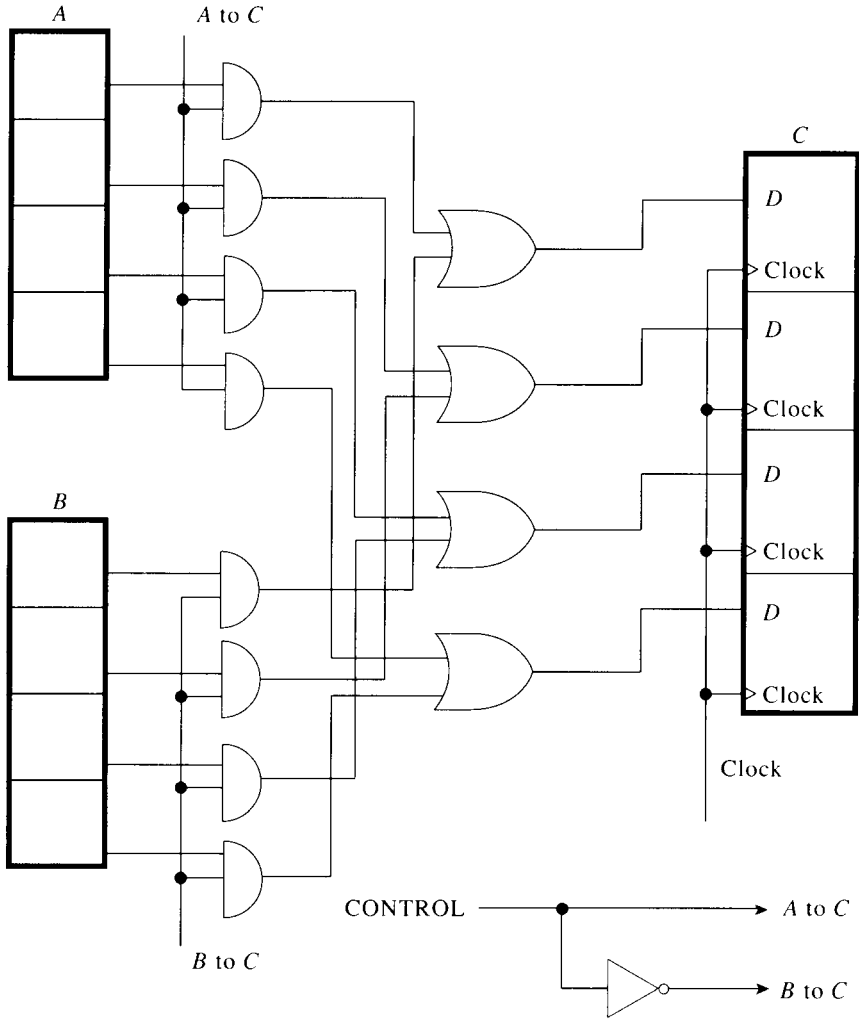



(b) Timing

**Figure 2.37** Serial transfer scheme

the timing diagram in (b), we need four clock pulses to complete the transfer, and the control signal $X$ must stay at 1 during all the four clock pulses.

All data processing done in the processing unit of a computer is accomplished by one or more register transfer operations. It is often



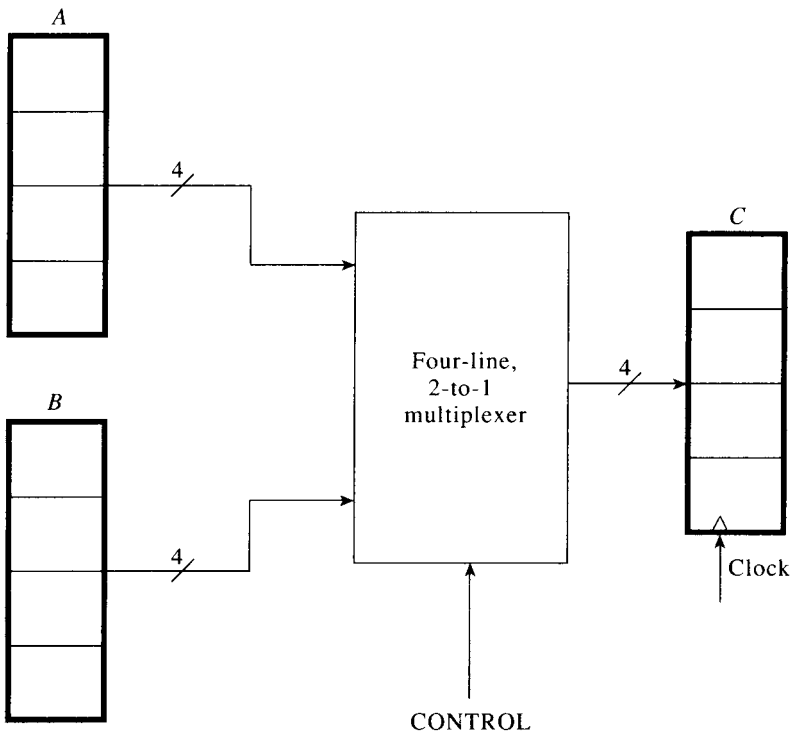

(a) Transfer by control signals

**Figure 2.38** Transfer from multiple-source registers

required that data in one register be transferred into several other registers or a register receive its inputs from one or more other registers. Figure 2.38 shows two schemes for transferring the contents of either register $A$ or register $B$ into register $C$. When the *control* signal "$A$ to $C$" is on, contents of $A$ are moved into $C$. When "$B$ to $C$" signal is on, contents of $B$ are moved into $C$. Only one control signal can be active at any time. This can be accomplished by using the true and complement of the same control signal to select one of the two transfer paths.

Note that it takes at least two gate delay times after the activation of the control signal for the data from either $A$ or $B$ to reach the inputs of $C$. The control signal must stay active during this time and until the occurrence of the rising edge of the clock pulse that gates the data into $C$.

Figure 2.38(b) shows the use of a 4-line 2-to-1 multiplexer to accomplish the register transfer required in (a).




(b) Transfer by multiplexer

**Figure 2.38** (Continued)

## 2.8 REGISTER TRANSFER SCHEMES

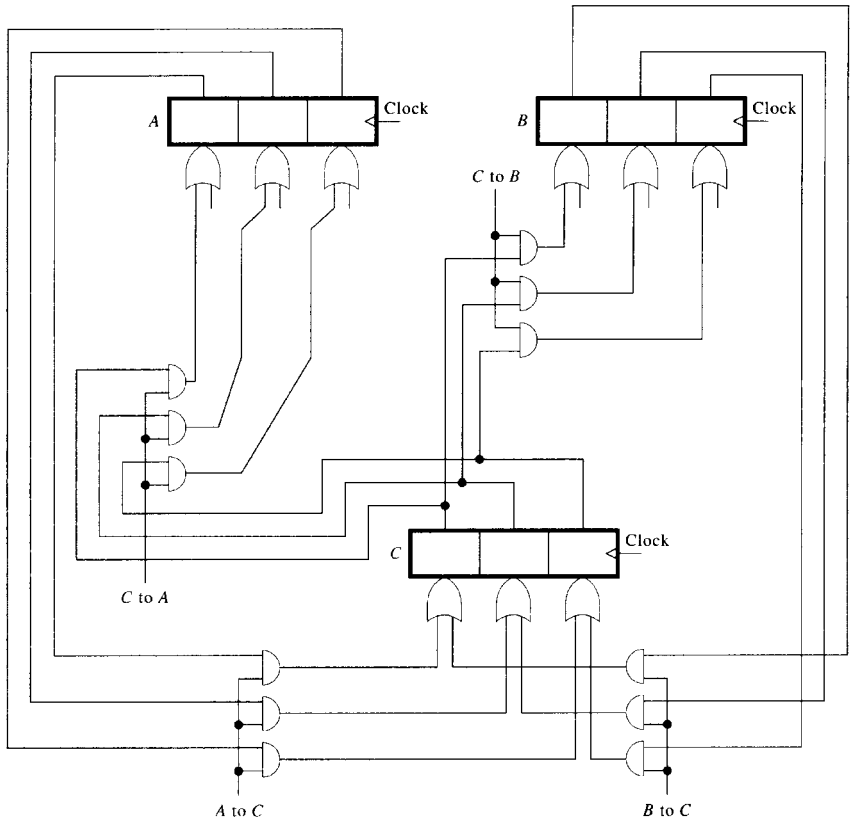
When it is required to transfer data between several registers to complete a processing sequence in a digital computer, one of two transfer schemes is generally used: (a) point-to-point and (2) bus. In a point-to-point scheme, there will be one transfer path between each of the two registers involved in the data transfer. In a bus scheme, one common path is time-shared for all register transfers.

### 2.8.1 Point-to-Point Transfer

The hardware required for a *point-to-point* transfer between three 3-bit registers $A$, $B$, and $C$ is shown in Fig. 2.39. Only a few of the paths are shown. "$A$ to $C$" and "$B$ to $C$" are control signals used to bring the data transfer. This scheme allows more than one transfer to be made at the same time (in parallel) because independent data paths are available. For example, the control signals "$A$ to $C$" and "$C$ to $B$" can both be enabled at the same time. The disadvantage of the scheme is that the amount of hardware required for the transfer increases rapidly as additional registers are included, and each new register is connected to other registers through newer data paths. This growth makes the scheme too expensive; hence, a point-to-point scheme is used only when fast, parallel operation is desired.

### 2.8.2 Bus Transfer

Figure 2.40(a) shows a *bus* scheme for the transfer of data between three 3-bit registers. A bus is a common data path (highway) that each register either feeds data into (i.e., contents of the register ON the bus) or takes data from (i.e., register OFF the bus). At any time, only one register can be putting data on the bus. This requires that bits in the same position in each register be ORed and connected to the corresponding bit (line) of the bus. Figure 2.40(b) shows typical timing for the transfer from $A$ to $C$. Control signals "$A$ to BUS" and "BUS to $C$" have to be 1 simultaneously for the transfer to take place. Several registers can receive data from the bus simultaneously, but only one register can put data on the bus at any time. Thus the bus transfer scheme is slower than the point-to-point scheme, but the hardware requirements are considerably less. Further, additional registers can be added to the bus structure just by adding two paths, one each from bus to register and register to bus. For these reasons, bus transfer is the most commonly used data transfer scheme.




Note: paths from $A$ to $B$ and $B$ to $A$ are not shown

**Figure 2.39** Point-to-point transfer

In practice, a large number of registers are connected to a bus. This requires the use of OR gates with many inputs to form the bus interconnection. Two special types of outputs available on certain gates permit an easier realization of the OR function: gates with "open-collector" output or "tri-state" output. Figure 2.41(a) shows 1 bit of a bus built using the *open-collector gates*. The outputs of these special gates can be tied together to provide the OR function. One other commonly used device, a *tristate gate*, is shown in Fig. 2.41(b). When the gate is enabled (enable = 1), the output is a function of the input; if disabled (enable = 0), the output is nonexistent electrically. The scheme shown in Fig. 2.41(a) realizes the OR function using tristate buffers.

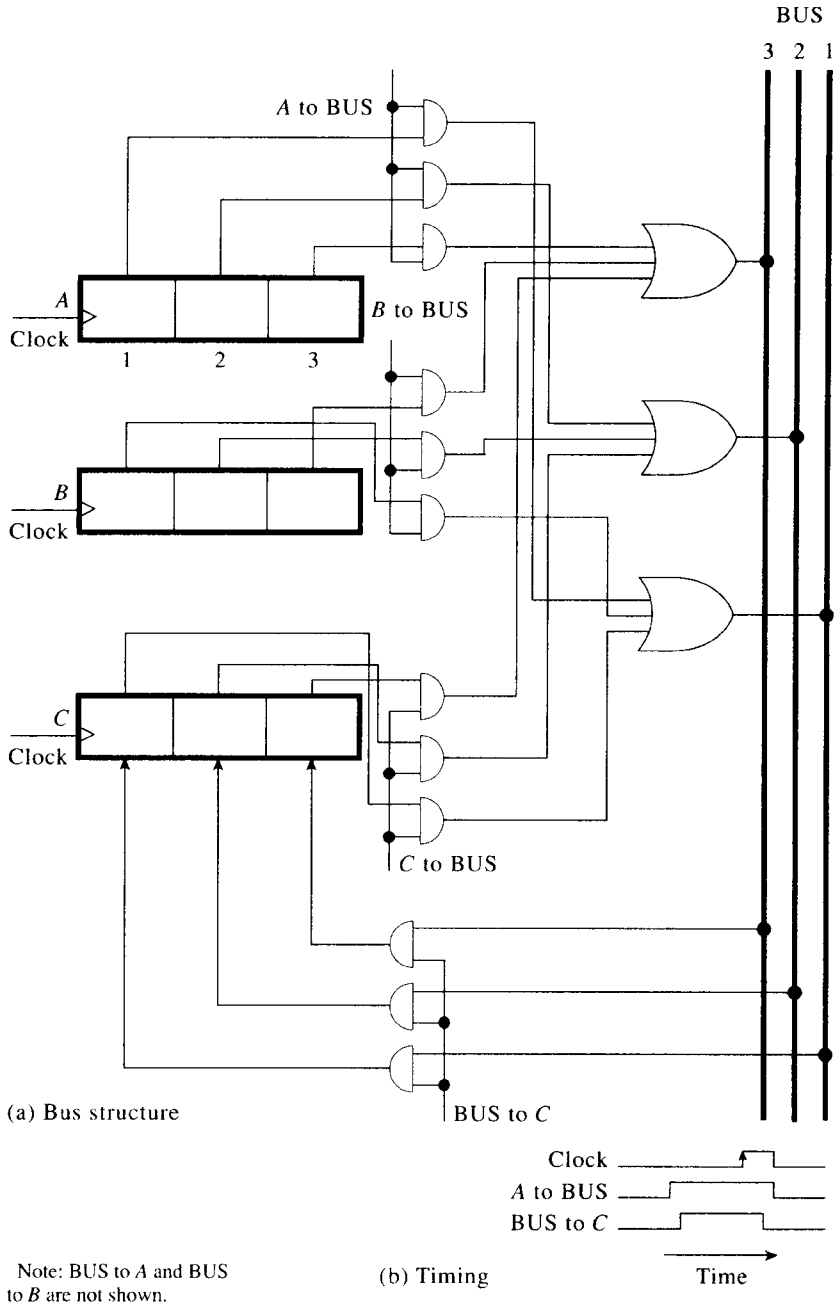
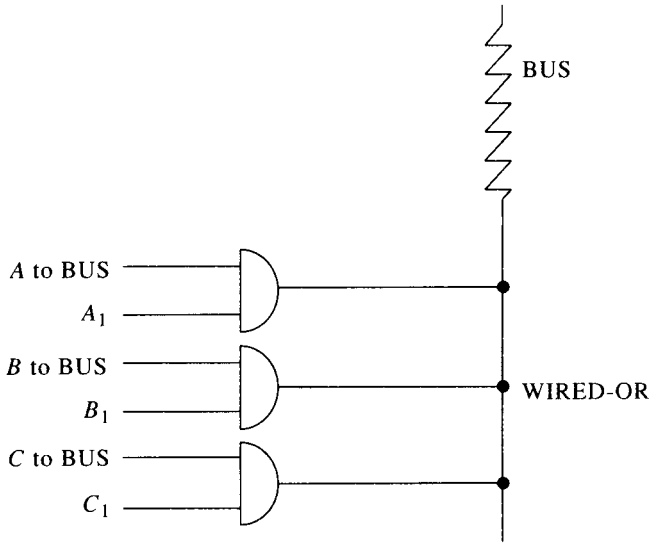
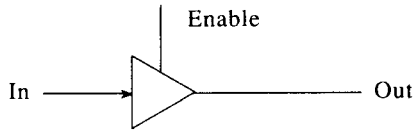


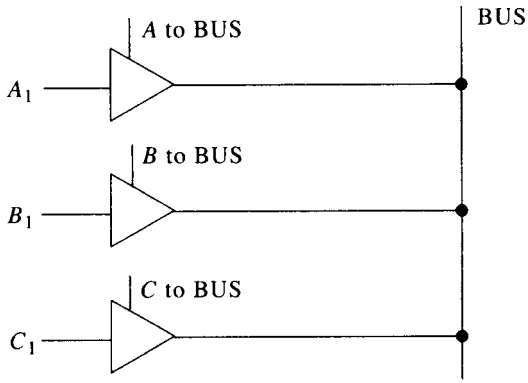

**Figure 2.40** Bus transfer




(a) Open-collector gates




(b) Tristate buffer




(c) OR using tristates

**Figure 2.41** Special devices for bus interface

Figure 2.42 shows the complete bus structure for transfer between four 4-bit registers $A$, $B$, $C$, and $D$. The SOURCE register is connected to the bus by enabling the appropriate tristate, as selected by the outputs of the source control 2-to-4 decoder. The DESTINATION register is selected by the outputs of the destination control 2-to-4 decoder. Note that a 4-line 4-to-1 multiplexer could also be used to form the connections from the registers to the bus.

## 2.9 REGISTER TRANSFER LANGUAGES

Since register transfer is the basic operation in a digital computer, several register transfer notations have evolved over the past decade. These notations, complete enough to describe any digital computer at the register transfer level, have come to be known as *register transfer languages*. Since they are used to describe the hardware structure and behavior of digital systems, they are more generally known as *hardware description languages* (HDL). For the purposes of this book, the details of a relatively simple HDL are shown here.

Tables 2.1 and 2.2 show the basic operators and constructs of our HDL. The general format of a register transfer is

Destination ← Source.

**Table 2.1** HDL Operators

| Operator | Description | Examples |
|---|---|---|
| Left arrow ← | Transfer operator | $Y \leftarrow X$. Contents of register $X$ are transferred to register $Y$. |
| Plus + | Addition | $Z \leftarrow X + Y$. |
| Minus − | Subtraction | $Z \leftarrow X - Y$. |
| ¢ | Concatenation | $C \leftarrow A \,¢\, B$. |
| Prime ′ | Complement | $D \leftarrow A'$. |
| ∧ | Logical AND | $C \leftarrow A \wedge B$. |
| ∨ | Logical OR | $C \leftarrow A \vee B$. |
| *SHL* | Shift left 1 bit; zero filled on right | $A \leftarrow SHL\ (A)$. |
| *SHR* | Shift right 1 bit; copy most significant bit on left | $A \leftarrow SHR\ (A)$. |

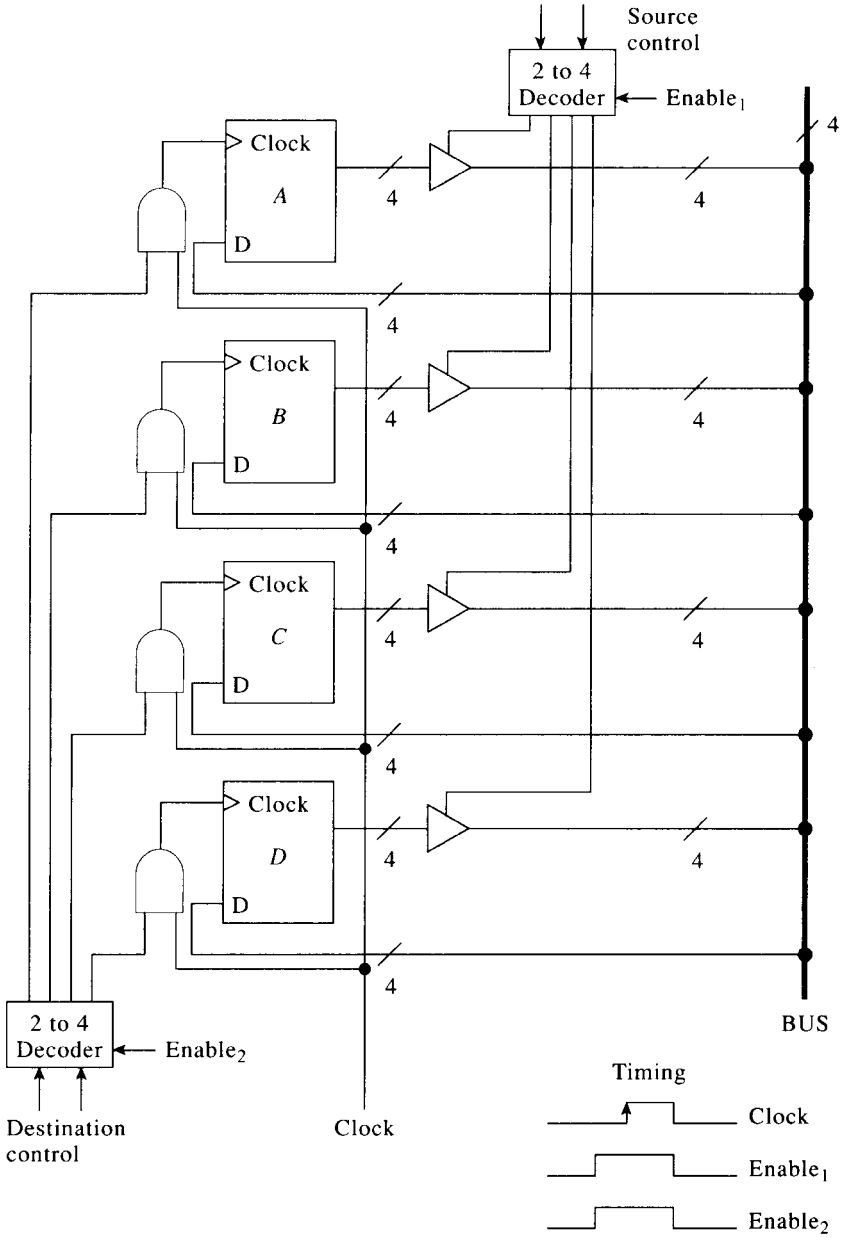



**Figure 2.42** Bus structure

**Table 2.2** HDL Constructs

| Construct | Description | Examples | |
|---|---|---|---|
| Capital-letter strings | Denote registers | *ACC*, *A*, *MBR* | |
| Subscripts | Denote a bit or a range of bits of a register | $A_0$, $A_{15}$ | Single bit. |
| | | $A_{5-15}$ | Bits are numbered left to right, bits 5 through 15. |
| | | $A_{5-0}$ | Bits are numbered right to left, bits 0 through 5. |
| Parentheses ( ) | Denote a portion of a register (subregister) | *IR (ADR)* | *ADR* portion of the register *IR*; this is a symbolic notation to address a range of bits. |
| Colon: | Serves as control function delimiter | *ADD:* | Terminates the control signal definition. |
| Comma, | Separates register transfers; implies that transfers are simultaneous | $Y \leftarrow X, Q \leftarrow P.$ | |
| Period | Terminates register transfer statement | $Y \leftarrow X.$ | |

where "Source" is a register or an expression consisting of registers and operators, and "Destination" is a register or a concatenation (linked series) of registers. The number of bits in source and destination must be equal. A period (".") terminates a register transfer statement.

A transfer controlled by a control signal has the format

Control: transfer.

Multiple transfers controlled by a control signal are indicated by

Control: $\text{transfer}_1, \text{transfer}_2, \cdots, \text{transfer}_n$.

The transfers are simultaneous.

The general format of a conditional register transfer is

IF condition THEN $\text{transfer}_1$
ELSE $\text{transfer}_2$.

where "condition" is a Boolean expression, "$\text{transfer}_1$" occurs if condition is TRUE (or 1), and "$\text{transfer}_2$" occurs if condition is FALSE (or 0).

The ELSE clause is optional. Thus,

IF condition THEN transfer.

is valid.

A control signal can be associated with a conditional register transfer:

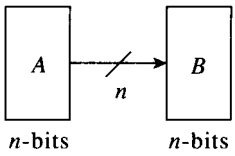
Control: IF condition THEN $\text{transfer}_1$
ELSE $\text{transfer}_2$.

Example 2.9 illustrates the features of the HDL.

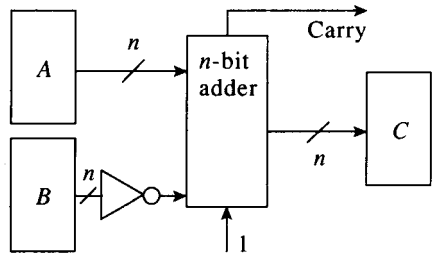
---

**Example 2.9**

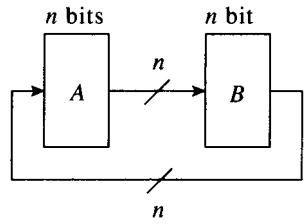
$B \leftarrow A$. $A$ and $B$ must have the same number of bits.



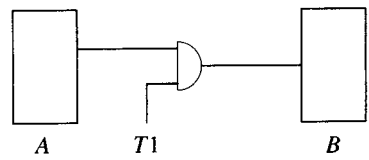

$C \leftarrow A + B' + 1$. 2s complement of $B$ added to $A$, transferred to $C$.



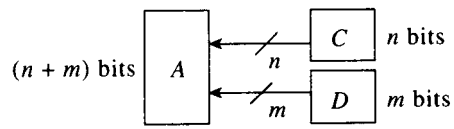

$B \leftarrow A, A \leftarrow B.$ $A$ and $B$ exchange. ($A$ and $B$ must be formed using master-slave flip-flops to accomplish this exchange.)



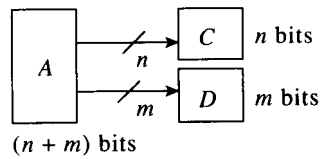

$T1 : B \leftarrow A.$



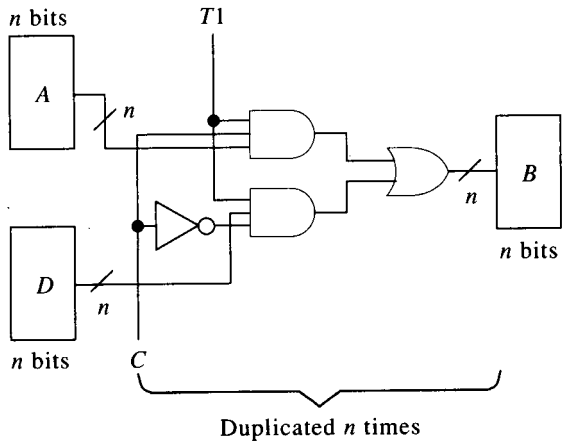

$A \leftarrow C \not\in D.$ The total number of bits in $C$ and $D$ must be equal to that in $A$.




$C \not\in D \leftarrow A.$ Reverse operation of the above.



$T1$: IF $C$ THEN $B \leftarrow A$ Equivalent to $T1 \wedge C : B \leftarrow A$ and $T1 \wedge C'$:
ELSE $B \leftarrow D$. $B \leftarrow D$




The transfers in Fig. 2.38 can be described by the statement

If control THEN $C \leftarrow A$
ELSE $C \leftarrow B$.

## 2.10 DESIGNING SEQUENTIAL CIRCUITS WITH INTEGRATED CIRCUITS

Appendix C shows some small- and medium-scale integrated circuits from the transistor–transistor logic (TTL) family. The reader is referred to IC manufacturer catalogs for further details on these ICs. A sequential circuit can be designed by following the classical design procedure described in this chapter. As a final step in the design, the circuit components (flip-flops, registers, etc.) are selected by referring to manufacturer catalogs.

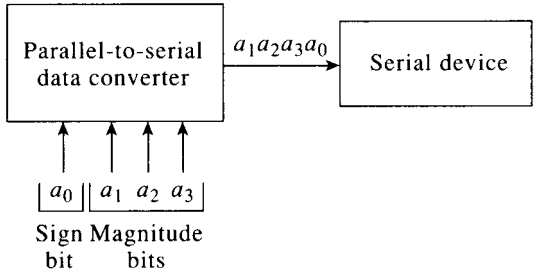
It is often possible to design sequential circuits without following the classical design procedure. The serial adder design (see Example 2.7) is one example. Where the number of states in practical circuits becomes so large that the classical design procedure becomes impractical, the circuit functions are usually partitioned and each partition is separately designed. Ad hoc methods of design based on familiarity with available ICs may be used in designing a partition or the complete circuit. Example 2.10 illustrates the design process using ICs.

**Example 2.10** *Parallel-to-serial data converter*. The object of our design is a parallel-to-serial data converter that accepts 4-bit data in parallel and produces as its output a serial-bit stream of the data input into it. The input consists of the sign bit ($a_0$) and three magnitude bits ($a_1a_2a_3$). The serial device expects to receive the sign bit $a_0$ first, followed by the three magnitude bits in the order $a_3a_2a_1$ as shown in Fig. 2.43(a).

Note that the output bit pattern can be obtained by circulating the input data right three times and then shifting right one bit at a time. To perform this, a 4-bit shift register that can be loaded in parallel and can be right-shifted is required. TTL 7495 is one such circuit. From the 7495 circuit diagram, it can be deduced that the "mode" input must be 1 for the parallel load operation and has to be 0 for serial input and right shift modes. The $D$ output must be connected to the "serial" input line, for circulating the data.

Figure 2.43(b) shows the details of the circuit operation. The complete operation needs eight steps, designated 0 through 7. Two more idle steps 8 and 9 are shown, since a decade counter (7490) is available that can count from 0 through 9. The circuit is shown in Fig. 2.43(c). The 4-bit output of the decade counter 7490 is decoded using a BCD-to-decimal decoder (7442). Since the outputs of 7442 are low-active, output 0 will have a value of 0 during the 0 time step and a value of 1 during other times. Hence, it can be used as mode control signal for 7495. Inputs $CP_1$ and $CP_2$ of 7495 must be tied together so that the circuit receives "clock" in both modes. Output 3 of 7442 is used to alert the serial device for data acceptance, starting at the next clock edge; output 8 indicates the idle state.

This example illustrates a simple design using ICs. In practice, the timing problems will be more severe. The triggering of flip-flops, data

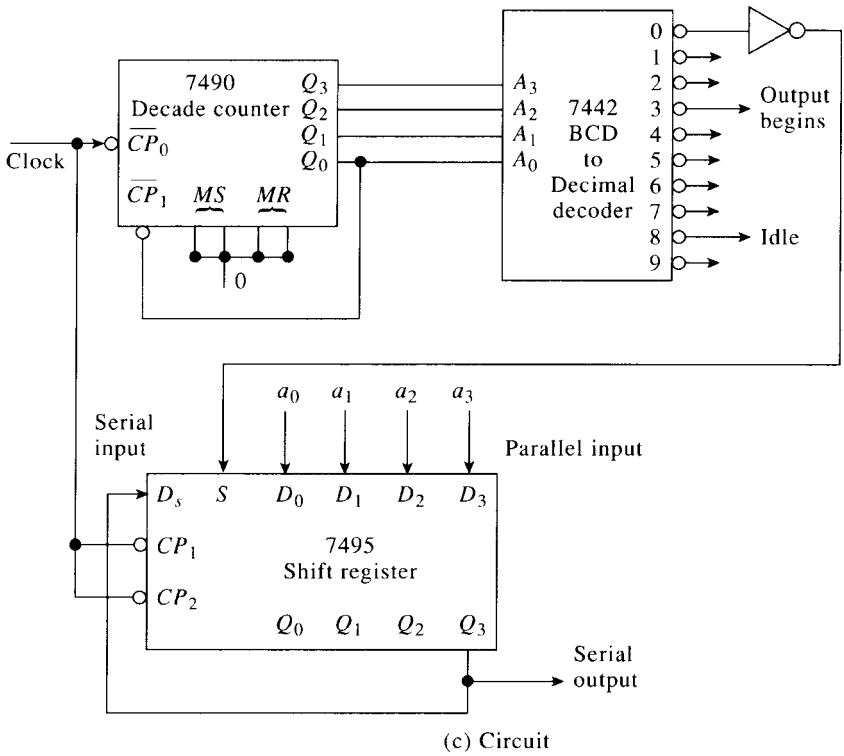



(a) Requirements

**Figure 2.43** A parallel-to-serial data converter

| Count | Event | Action | Mode |
|---|---|---|---|
| 0 | Load the register (Parallel) | Parallel in | 1 |
| 1 | ↑ | Shift circular | 0 |
| 2 | Circulate | " | 0 |
| 3 | ↓ | " | 0 |
| 4 | ↑ | Shift right | 0 |
| 5 | Serial | " | 0 |
| 6 | Output | " | 0 |
| 7 | ↓ | " | 0 |
| 8 | Idle | Idle | *d* |
| 9 | Idle | Idle | *d* |

(b) Operation details




(c) Circuit

**Figure 2.43** (Continued)

setup times, clock skews (i.e., arrival of clock on parallel lines at slightly different times due to differences in path delays), and other timing elements must be considered in detail.

## 2.11 SUMMARY

The analysis and design of synchronous sequential circuits described in this chapter are given as an overview of the subject. The reader is referred to the logic design texts listed at the end of this chapter for further details on these topics and also on asynchronous circuit analysis and design. IC manufacturer catalogs are an important source of information for logic designers, although the detailed electrical characteristics given in these catalogs are not required for this purposes of this book. Register transfer logic concepts described in this chapter will be used extensively in Chapter 5 in the logical design of a simple computer.

## REFERENCES

Blakeslee, T. R. *Digital Design with Standard MSI and LSI*, New York, NY: John Wiley, 1975.

*FAST TTL Logic Series Data Handbook*, Sunnyvale, CA: Phillips Semiconductors, 1992.

Greenfield, J. D. *Practical Design Using ICs*, New York, NY: John Wiley, 1983.

Kohavi, Z. *Switching and Automata Theory*, New York, NY: McGraw-Hill, 1978.

Mano, M. M. *Digital Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

McCluskey, E. J. *Introduction to the Theory of Switching Circuits*, New York, NY: McGraw-Hill, 1965.

*National Advanced Bipolar Logic Databook*, Santa Clara, CA: National Semiconductor Corporation, 1995.

Perry, D. L. *VHDL*, New York, NY: McGraw-Hill, 1991.

Shiva, S. G. *Introduction to Logic Design*, New York, NY: Marcel Dekker, 1998.

Thomas, D. E. and Moorby, P. R. *The VERILOG Hardware Description Language*, Boston, MA: Kluwer, 1998.

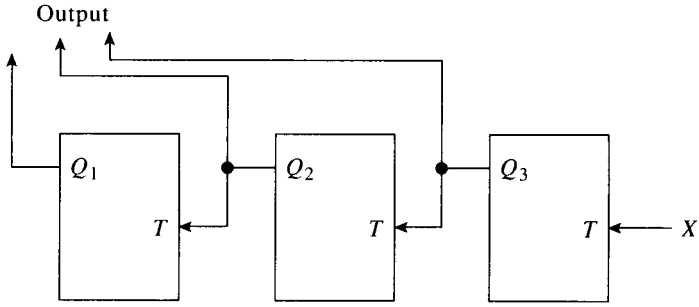
*TTL Data Manual*, Sunnyvale, CA: Signetics, 1987.

## PROBLEMS

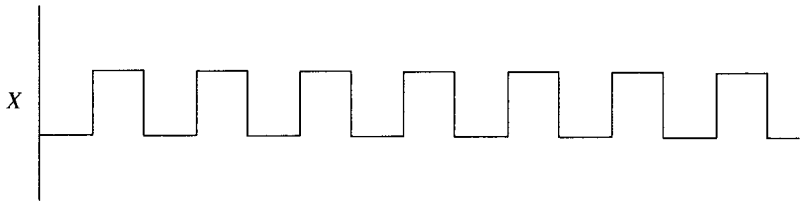
2.1 You are given a *JK* flip-flop. Design the circuitry around it to convert it into a (a) *T* flip-flop, (b) D flip-flop, and (c) *SR* flip-flop. Hint: A flip-flop is a sequential circuit. Start the design with the state table of the required flip-flop. Use a *JK* flip-flop in the design.

2.2 A *set-dominate flip-flop* is similar to an *SR* flip-flop, except that an input $S = R = 1$ will result in setting the flip-flop. Draw the state table and excitation table for the flip-flop.

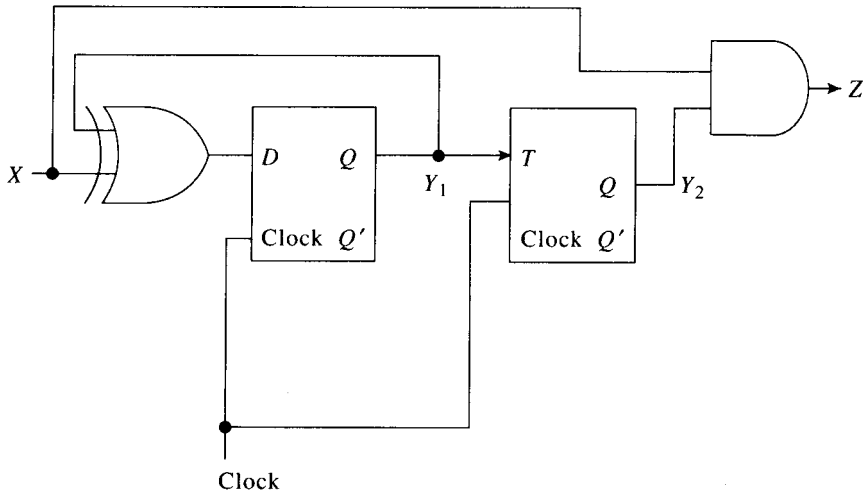
2.3 In the circuit shown below, note that the flip-flops are not clocked but are triggered by the falling edge of the *T* input.




Assume that the *X* input makes the transitions shown below, and complete the timing diagram showing the signals *Q*1, *Q*2 and *Q*3. (NOTE: This is a *ripple* counter.)

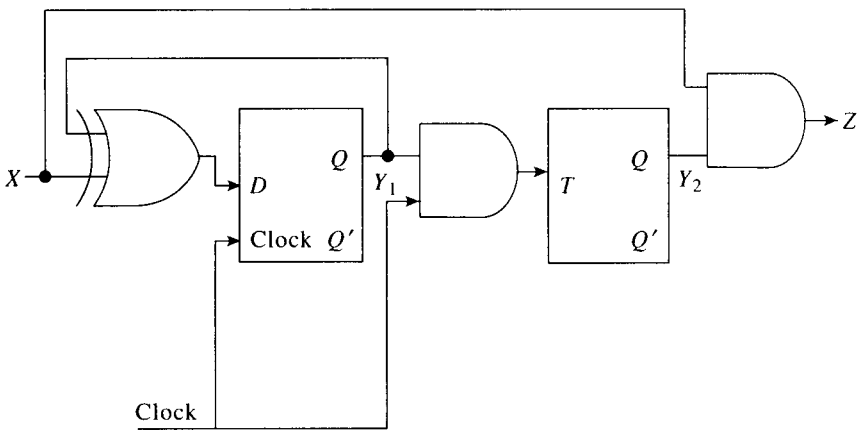


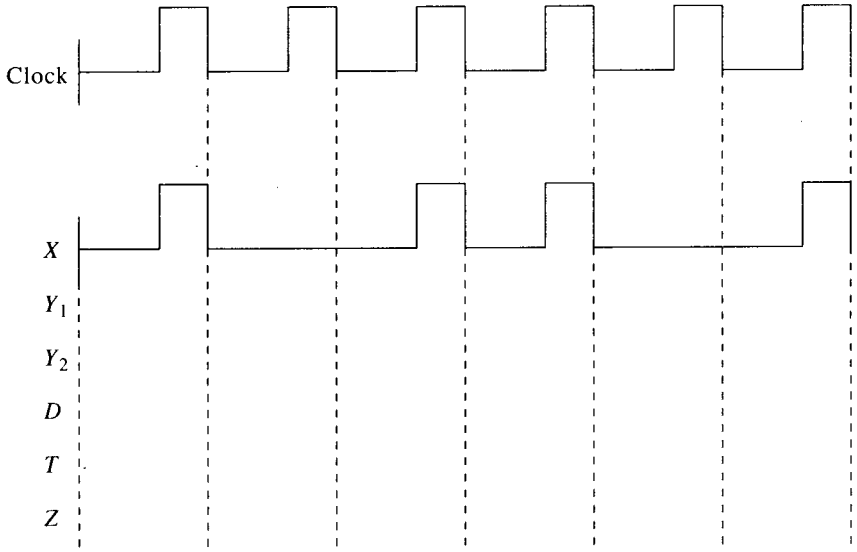
2.4 For the following circuit, derive the state table. Use the assignments $Y_1 Y_2$: $00 = A$, $01 = B$, $10 = C$, and $11 = D$.



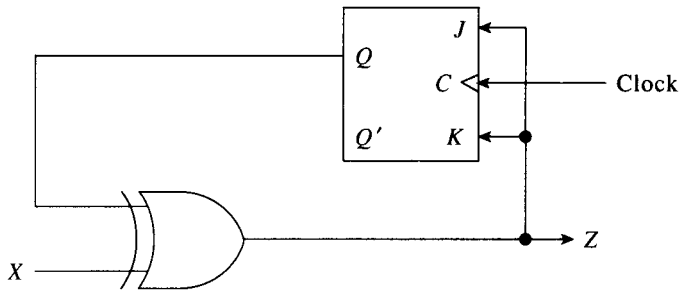

2.5 For the following circuit,

a. Complete the timing diagram starting with $Y_1 Y_2 = 00$ (at time = 0).
b. Derive excitation tables.
c. Derive the state table, using $Y_1 Y_2$: $00 = A$, $01 = B$, $11 = C$, and $10 = D$. Assume that the flip-flops are triggered by the raising edge of the clock.






2.6 The circuit shown below gave an output sequence of $Z = 11011111$ for an input sequence $X = 01101010$. What was the starting state?

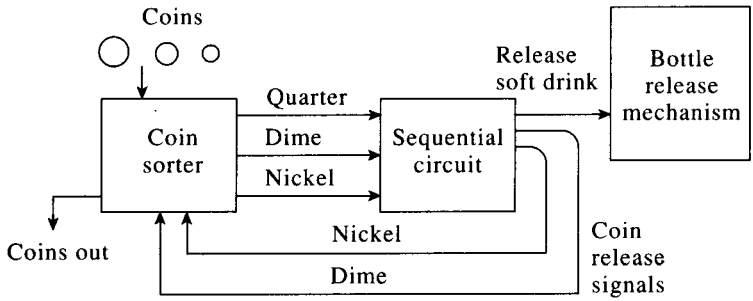



2.7 Construct state diagrams for sequence detectors that can detect the following sequences:

a. 11011 (sequences may overlap)
b. 11011 (overlap not allowed)
c. 1101 or 1001 (overlap allowed)
d. 1101 or 1001 (overlap not allowed)

2.8 Implement the state diagrams from Problem 2.7 using *JK* flip-flops.

2.9 Design the circuit for a soft drink machine. Each bottle costs thirty cents. The machine accepts nickels, dimes, and quarters. Assume a coin sorter that accepts coins and provides three signals, one each for the three types of coins. Assume that the signals on these three lines are separated far enough so that the other circuits can make a state transition between pulses received on any two of these lines. Correct change must be released. The sequential circuit must generate signals to release correct change (one signal for each type of coin) and the soft drink bottle. Use *JK* flip-flops in your design.




(The coin sorter produces a pulse on its appropriate output line for each coin it receives. It releases an appropriate coin when it receives a coin release signal.)

2.10 Derive the state diagram for an odd parity checker. The input arrives on a single input line X, one bit at a time. The circuit should produce an output of 1 if the number of 1s in the input sequence of four bits is odd. The circuit should be reset to the starting state after every 4-bit sequence on the input.

2.11 There are two 4-bit registers $A$ and $B$, built out of $SR$ flip-flops. There is a control signal $C$. The following operations are needed:

If $C = 0$, send contents of $A$ to $B$.

If $C = 1$, send 1s complement of contents of $A$ into $B$.

Draw the circuit to perform these functions

a. in parallel mode.

b. in serial mode.

2.12 Design the circuit in Problem 2.11a for a 2s complement transfer.

2.13 There are three 2-bit registers $A$, $B$, and $C$. Design the logic to perform:

AND: $C \leftarrow A \wedge B$.

OR: $C \leftarrow A \vee B$.

AND and OR are control signals. Each bit in the register is a $D$ flip-flop.

2.14 In your design for Problem 2.13, what happens if both the control signals are at 0 and a clock pulse comes along? Redesign the circuit (if necessary) to prevent the clearing of register $C$ under the above conditions. (HINT: Feed the output of $C$ to its input. What type of flip-flop should $C$ be made of?)

2.15 Implement the circuit of Problem 2.14 using appropriate multiplexers.

2.16 A 2-bit counter $C$ controls the register transfers shown below:

$C = 0$: B$\leftarrow A$. $\quad C = 2$: $B \leftarrow A + B$.
$C = 1$: $B \leftarrow A'$. $\quad C = 3$: $B \leftarrow 0$.

$A$ and $B$ are 2-bit registers. Draw the circuit. Use 4-to-1 multiplexers in your design. Show the details of register $B$.

2.17 Draw a bus structure to perform the operations in Problem 2.16

2.18 Connect four 5-bit registers $A$, $B$, $C$, and $D$ using a bus structure capable of performing the following:

$C0$: $B \leftarrow A$. $\quad C4$: $A \leftarrow C + D$.
$C1$: $C \leftarrow A \wedge B$. $\quad C5$: $B \leftarrow C' \wedge D'$.
$C2$: $D \leftarrow A \vee B$. $\quad C6$: $D \leftarrow A + C$.
$C3$: $A \leftarrow A + B'$. $\quad C7$: $B \leftarrow A + C'$.

$C0$ through $C7$ are control signals. Use a 3-bit counter and a decoder to generate those signals. Assume tristate outputs for each register.

2.19 Assume regular outputs for each register in Problem 2.18. How many OR gates are needed to implement the bus structure?

2.20 Design a 4-bit register using TTL 7474 D flip-flops. Include a LOAD control input. The data should enter the register when LOAD is high and at (a) positive edge of the clock and (b) negative edge of the clock.

2.21 Compare the complexity of the circuit in Problem 2.20 with that of the TTL 74173.

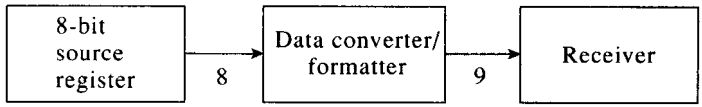
2.22 Design an 8-bit shift register using 74109 JK flip-flops. Parallel and serial inputs, serial output, and right-shift features are required.

2.23 Compare the complexity of the circuit in Problem 2.22 with that of the TTL 74199.

2.24 To add two positive numbers $A$ and $B$, $A$ can be incremented by 1, $B$ times. Design the circuit to perform the addition of two 4-bit numbers $A$ and $B$. $A$ and $B$ registers could each be an up/down counter. You have to stop incrementing $A$ when $B$ reaches 0. Assume $B$ is positive.

2.25 Repeat Problem 2.24 for $B$, a negative number, represented in 2s complement system.

2.26 It is required to transmit data on a serial line. The data are in an 8-bit register. The receiver expects the 8-bit data followed by a parity bit. Odd parity is used; i.e., if the number of 1s in the data is even, the parity bit will be 1, otherwise 0.




Design the data converter/formatter circuit. Use existing ICs from a catalog, if possible.

# 3
# Memory and Storage

We have demonstrated the use of flip-flops in storing binary information. Several flip-flops put together form a register. A register is used either to store data temporarily or to manipulate data stored in it using the logic circuitry around it. The *memory* subsystem of a digital computer is functionally a set of such registers where data and programs are stored. The instructions from the programs stored in memory are retrieved by the control unit of the machine (digital computer system) and are decoded to perform the appropriate operation on the data stored either in memory or in a set of registers in the processing unit.

For optimum operation of the machine, it is required that programs and data be accessible by control and processing units as quickly as possible. The *main memory* (primary memory) allows such a fast access. This fast-access requirement adds a considerable amount of hardware to the main memory and thus makes it expensive. To reduce memory cost, data and programs not immediately needed by the machine are normally stored in a less expensive *secondary memory* subsystem. They are brought into the main memory as the processing unit needs them. The larger the main memory, the more information it can store and hence the faster the processing, since most of the information required is immediately available. But because main-memory hardware is expensive, a speed-cost tradeoff is needed to decide on the amounts of main and secondary storage needed. This chapter provides models of operation for the most commonly used types of memories, followed by a brief description of memory devices and organization. Chapter 8 expands on the memory system design discussed here and covers *virtual* and *cache* memory schemes.

We will provide the models of operation for the four most commonly used types of memories in the next section. This section is simply a functional description of these memory systems and hence does not cover the hardware level details. Section 3.2 lists the parameters used in evaluating

memory systems and describes the memory hierarchy in computer systems. Memory system parameters such as cost, density, and so on change so rapidly that a listing of such characteristics becomes obsolete before it can be published in a book. The magazines listed at the end of the chapter should be consulted for such details. Section 3.3 describes the popular semiconductor memory devices and the design of primary memory system using these devices in detail, followed by a brief description of popular secondary memory devices. Representative memory ICs are briefly described in Appendix C, and the design of primary memory using these ICs is described in Section 3.4. The description of commercial memory ICs provided here highlights the important characteristics only and is by no means a substitute for manufacturers' manuals.

## 3.1 TYPES OF MEMORY

Depending on the mechanism used to store and retrieve data, a memory system can be classified as one of the following four types:

1. Random-access memory (RAM):
   a. Read/write memory (RWM);
   b. Read-only memory (ROM).
2. Content-addressable memory (CAM) or associative memory (AM)
3. Sequential-access memory (SAM).
4. Direct-access memory (DAM).

Primary memory is of the RAM type. CAMs are used in special applications in which rapid data search and retrieval are needed. SAM and DAM are used as secondary memory devices.

### 3.1.1 Random-access Memory

In a RAM, any addressable location in the memory can be accessed in a random manner. That is, the process of reading from and writing into a location in a RAM is the same and consumes an equal amount of time no matter where the location is physically in the memory. The two types of RAM available are read/write and read-only memories.

## Read/Write Memory

The most common type of main memory is the read/write memory (RWM), whose model is shown in Fig. 3.1. In an RWM, each memory register or memory location has an "address" associated with it. Data are input into (written into) and output from (read from) a memory location by accessing the location using its "address". The memory address register (MAR) of Fig. 3.1 stores such an address. With $n$ bits in the MAR, $2^n$ locations can be addressed, and they are numbered from 0 through $2^n - 1$.

Transfer of data in and out of memory is usually in terms of a set of bits known as a *memory word*. Each of the $2^n$ words in the memory of Fig. 3.1 has $m$ bits. Thus, this is a $(2^n \times m)$-bit memory. This is a common notation used to describe random-access memories. In general, an $(N \times M)$-unit memory contains $N$ words of $M$ units each. A "unit" is either a bit, a byte (8 bits), or a word of certain number of bits. A memory buffer register (MBR) is used to store the data to be written into or read from a memory word. To read the memory, the address of the memory word to be read from is provided in

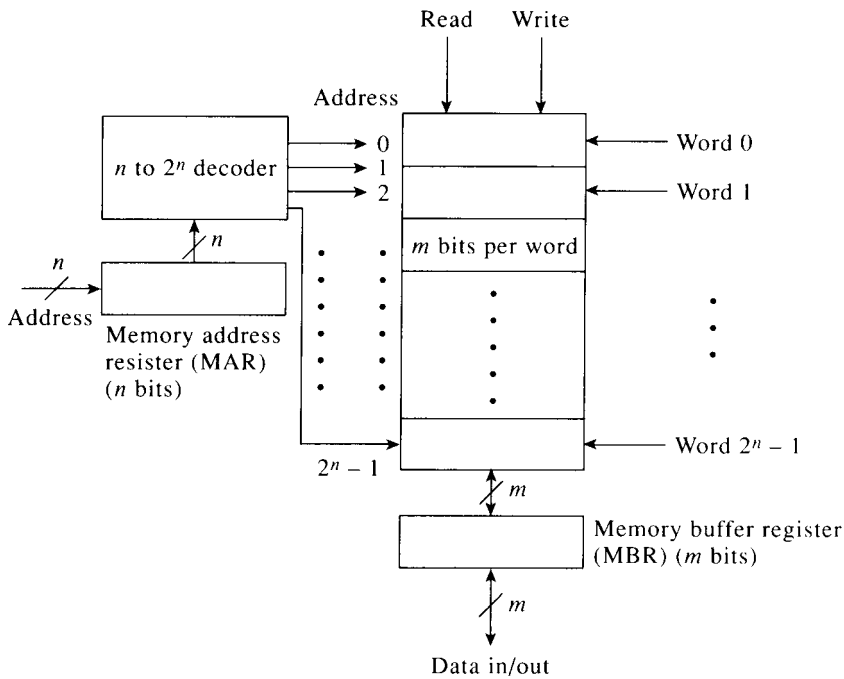



**Figure 3.1** Read/write memory

MAR and the Read signal is set to 1. A copy of the contents of the addressed memory word is then brought by the memory logic into the MBR. The content of the memory word is thus not altered by a read operation. To write a word into the memory, the data to be written are placed in MBR by external logic; the address of the location into which the data are to be written is placed in MAR; and the Write signal is set to 1. The memory logic then transfers the MBR content into the addressed memory location. The content of the memory word is thus altered during a write operation.

A memory word is defined as the most often accessed unit of data. The typical word sizes used in memory organizations of commercially available machines are 6, 16, 32, 36, and 64 bits. In addition to addressing a memory word, it is possible to address a portion of it (e.g., half-word, quarter-word) or a multiple of it (e.g., double word, quad word), depending on the memory organization. In a "byte-addressable," memory, for example, an address is associated with each byte (usually eight bits per byte) in the memory, and a memory word consists of one or more bytes.

The literature routinely uses the acronym RAM to mean RWM. We will follow this popular practice and use RWM only when the context requires us to be more specific. We have included MAR and MBR as components of the memory system in this model. In practice, these registers may not be located in the memory subsystem, but other registers in the system may serve the functions of these registers.

#### Read-Only Memory

Read-only memory (ROM) is also a random-access memory, except that data can only be read from it. Data are usually written into a ROM either by the memory manufacturer or by the user in an off-line mode; i.e., by special devices that can write (burn) the data pattern into the ROM. A model of ROM is shown in Fig. 3.2. A ROM is also used as main memory and contains data and programs that are not usually altered in real time during the system operation. The memory buffer register is not shown in Fig. 3.2. In general, we assume that the data on output lines are available as long as the memory enable signal is on and it is latched into an external buffer register. A buffer is provided as part of the memory system, in some technologies.

### 3.1.2 Content-addressable Memory

In this type of memory, the concept of address is not usually present: rather, the memory logic searches for the locations containing a specific pattern,

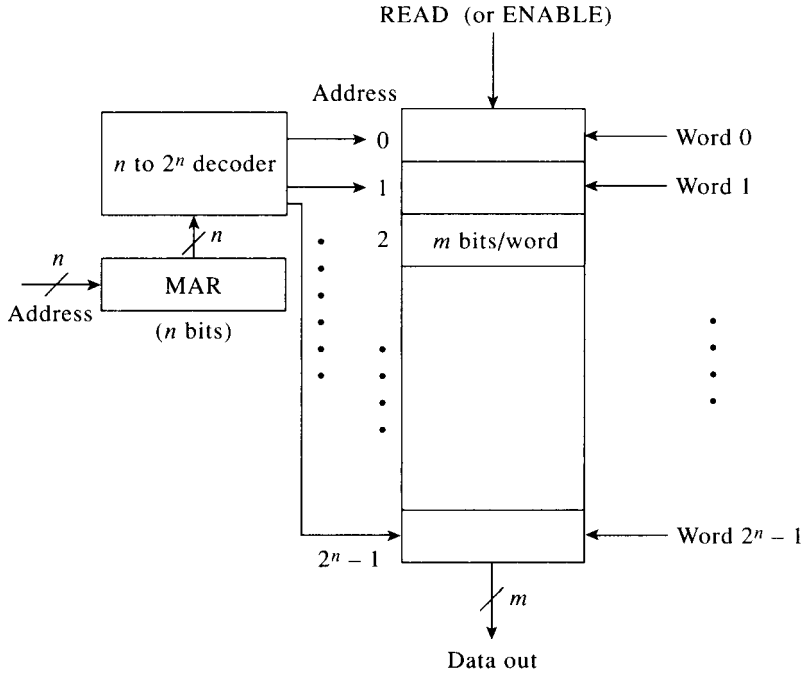



**Figure 3.2** Read-only memory

and hence the descriptor "content addressable" or "associative" is used. In the typical operation of this memory, the data to be searched for are first provided to the memory. The memory hardware then searches for a match and either identifies the location or locations containing that data or returns with a "no match" if none of the locations contain the data.

A mode for an *associative memory* is shown in Fig. 3.3. The data to be searched for are first placed in the *data register*. The data need not occupy the complete data register. The *mask register* is used to identify the region of the data register that is of interest for the particular search. Typically, corresponding mask register bits are set to 1. The *word-select register* bits are set to indicate only those words that are to be involved in the search. The memory hardware then searches through those words in only those bit positions in which the mask register bits are set. If the data thus selected match the content of the data register, the corresponding *results register* bit is set to 1. Depending on the application, all words responding to the search may be involved in further processing or one of the respondents may be selected. The *multiple-match resolver* (MMR) circuit implements this selection process.

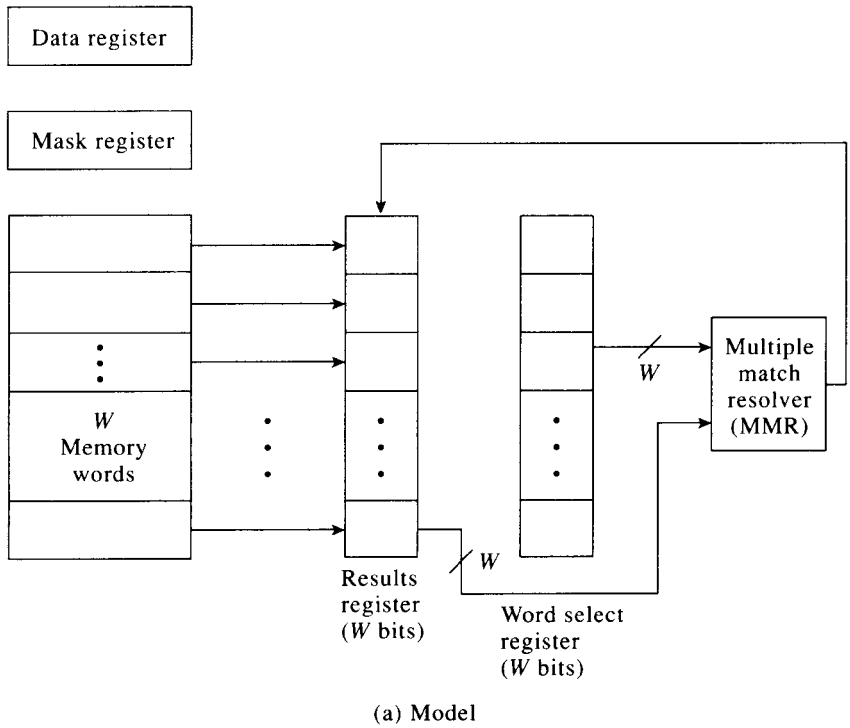
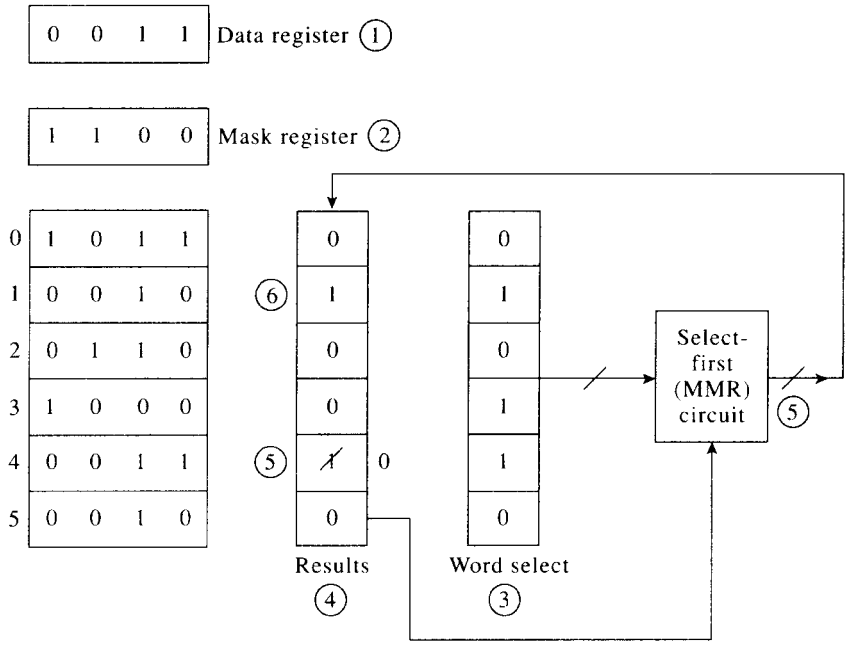



**Figure 3.3** Associative memory

Note that the data matching operation is performed in parallel. Hence, extensive hardware is needed to implement this memory. Figure 3.3(b) shows the sequence of search operation. A "select first" MMR circuit is used here to select the first respondent among all the respondent words for further processing.

Associative memories are useful when an identical operation must be performed on several pieces of data simultaneously or when a particular data pattern must be searched for in parallel. For example, if each memory word is a record in a personnel file, the records corresponding to the set of female employees 25 years old can be searched for by setting the data and mask register bits appropriately. If sufficient logic is provided, all records responding to the search can also be updated simultaneously.

In practice, content-addressable memories (CAMs) are built out of RAM components and as such have the same addressing capability. In




(b) Operation

1. The data being searched for is 0011.
2. The most significant two bits of mask register are 1s. Hence, only the corresponding two bits of data are compared with those of memory words 0 through 5.
3. The word select register bit setting indicates that only words 1, 3, and 4 are to be involved in the search process.
4. The results register indicates that words 1 and 4 have the needed data.
5. The select-first (MMR) circuit resets the results register bit, corresponding to word 4.
6. Word 1 is the final respondent.

This information can be used for updating word 1 contents.

Note: Comparison of the data register with memory words is done in parallel. The addresses shown (0 through 5) are for reference only

**Figure 3.3** (Continued)

fact, the MMR returns the address of the responding word or words in response to a search. The major application of CAM is for storing data on which rapid search and update operations are performed. The virtual memory scheme described in Chapter 8 shows an application for CAMs. We will return to the description of CAMs in section 3.3.2, where the detailed design of a CAM system is given.

### 3.1.3 Sequential-access Memory

A serial-input/serial-output shift register is the simplest model of sequential memory. In the right shift register of Fig. 3.4(a), data enter from the left input and leave the shift register from the right output. Because these are the only input and output available on the device, the data must be written into and read from the device in the sequence in which they are stored in the register. That is, every data item in sequence (from the first data item until the desired item) must be accessed in order to retrieve the required data. Thus, it is a SAM. In particular, this model corresponds to a *first-in/first-out* (FIFO) SAM, since the data items are retrieved in the order in which they were entered. This organization of a SAM is also called a *queue*. Note that in addition to the right-shift shift register, mechanisms for input and output of data (similar to MAR and MBR in the RAM model) are needed to build a FIFO memory.

Figure 3.4(b) shows the model for a *last-in/first-out* (LIFO) SAM. Here, a shift register that can shift both right and left is used. Data always enter through the left input and leave the register through the left output. To write, the data are placed on the input and the register is shifted right. While reading, the data on the output are read and the register is shifted left, thereby moving each item in the register left and presenting the next data item at the output. Note that the data are accessed from this device in a LIFO manner. This organization of a SAM is also called a *stack*. The data input operation is described as PUSHing the data into the stack and data are retrieved by POPing the stack.

Figure 3.5 shows FIFO and LIFO organizations for four-bit data words using eight-bit shift registers. Each of these SAM devices thus can store eight four-bit words.

Figure 3.6 shows the generalized model of a sequential-access storage system. Here, the read/write transducers read data from or write data onto the data storage medium at its current position. The medium is then moved

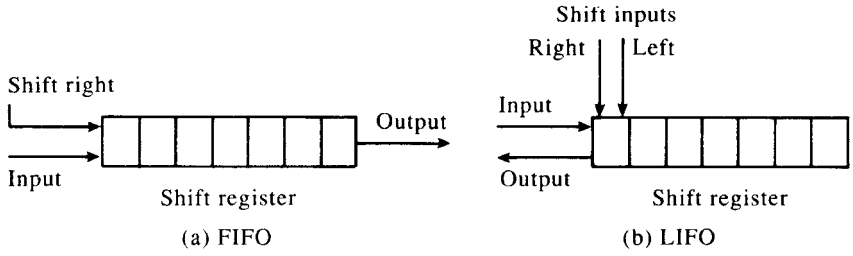
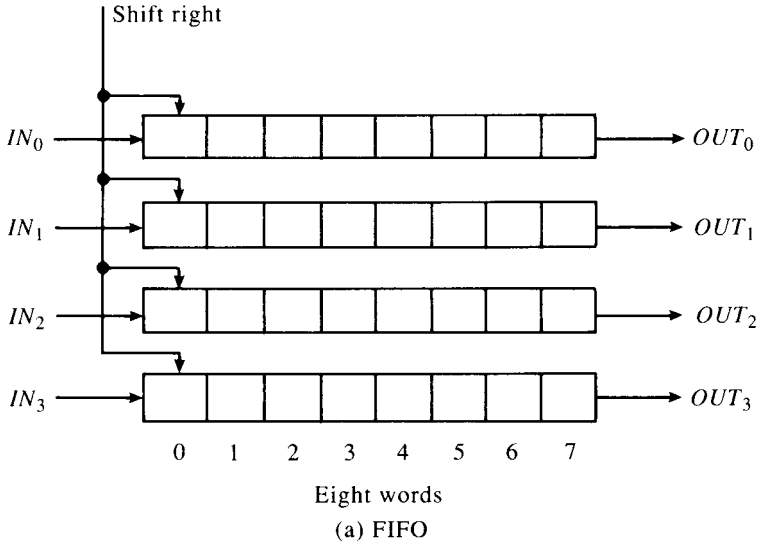



**Figure 3.4** Sequential-access device



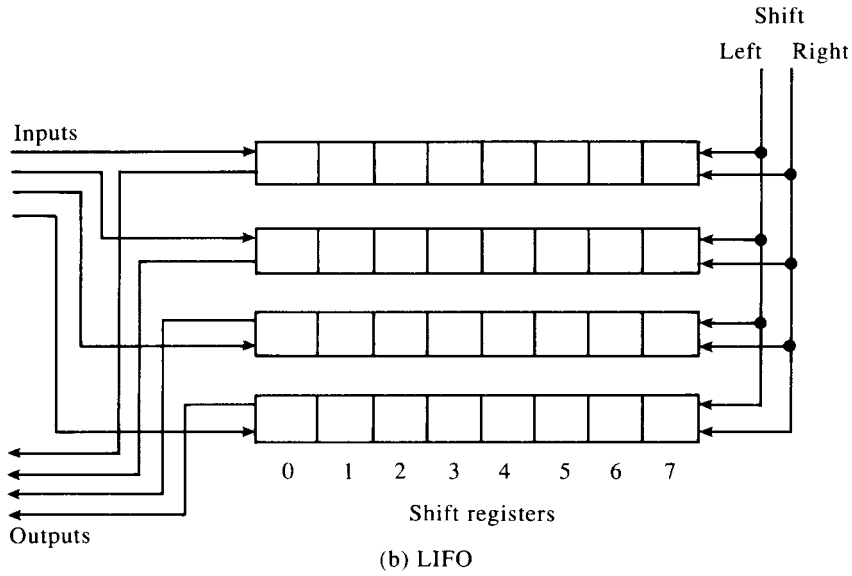




**Figure 3.5** Eight 4-bit word sequential-access device using shift registers

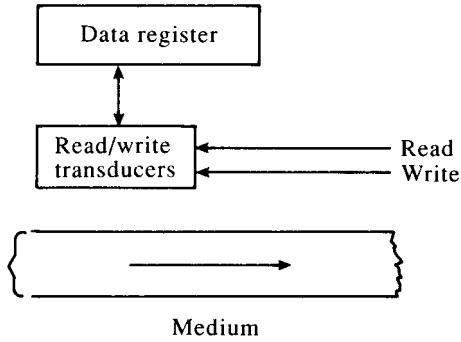



**Figure 3.6** Sequential-access storage

to the next position. Thus, each data item must be examined in sequence to retrieve the desired data. This model is applicable to secondary storage devices such as magnetic tape.

### 3.1.4 Direct-access Memory

Figure 3.7 shows the model of a DAM device (a magnetic disk) in which data are accessed in two steps:

1. The transducers move to a particular position determined by the addressing mechanism (cylinder, track).
2. The data on the selected track are accessed sequentially until the desired data are found.

This type of memory is used for secondary storage. It is also called *semirandom-access* memory, since the positioning of read/write transducers to the selected cylinder is random, and only the accessing of data within the selected track is sequential.

## 3.2 MEMORY SYSTEM PARAMETERS

The most important characteristics of any memory system are its capacity, data access time, the data transfer rate, the frequency at which memory can be accessed (the cycle time), and cost.

The *capacity* of the storage system is the maximum number of units (bits, bytes, or words) of data it can store. The capacity of a RAM, for instance, is the product of the number of memory words and the word size.

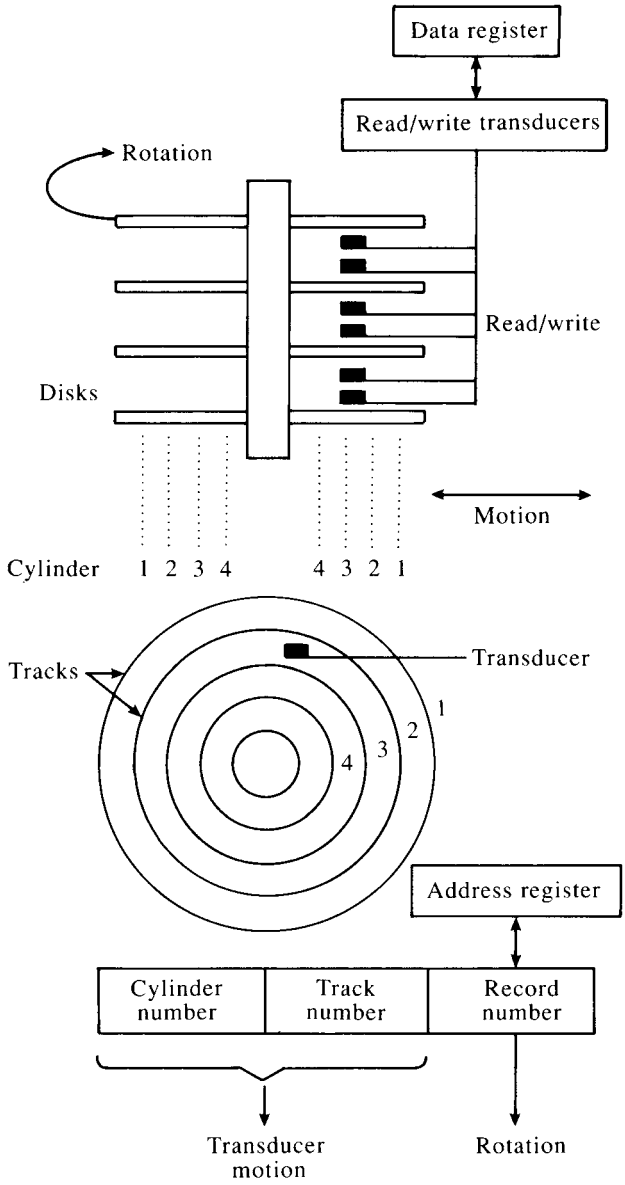



**Figure 3.7** Direct-access storage

A 2K × 4 memory, for example, can store 2K (K = 1024 = $2^{10}$) words each containing four bits, or a total of 2 × 1024 × 4 bits.

The *access time* is the time taken by the memory module to access the data after an address is provided to the module. The data appear in the MBR at the end of this time in a RAM. The access time in a non-random-access memory is a function of the location of the data on the medium with reference to the position of read/write transducers.

The *data transfer rate* is the number of bits per second at which the data can be read out of the memory. This rate is the product of the reciprocal of access time and the number of bits in the unit of data (data word) being read. This parameter is of more significance in nonrandom-access memory systems than in RAMs.

The *cycle time* is a measure of how often the memory can be accessed. The cycle time is equal to the access time in nondestructive readout memories in which the data can be read without being destroyed. In some storage systems, data are destroyed during a read operation (destructive read-out). A rewrite operation is necessary to restore the data. The cycle time in such devices is defined as the time it takes to read and restore data, since a new read operation cannot be performed until the rewrite has been completed.

The *cost* is the product of capacity and the price of memory device per bit. RAMs are usually more costly than other memory devices.

The *primary memory* of a computer system is always built out of RAM devices, thereby allowing the processing unit to access data and instructions in the memory as quickly as possible. It is necessary that the program or data be in the primary memory when the processing unit needs them. This would call for a large primary memory when programs and data blocks are large, thereby increasing the memory cost. In practice, it is not really necessary to store the complete program or data in the primary memory as long as the portion of the progrm or data needed by the processing unit is in the primary memory.

A *secondary memory* built out of direct or serial access devices is then used to store programs and data not immediately needed by the processing unit. Since random-access devices are more expensive than secondary memory devices, a cost-effective memory system results when the primary memory capacity is minimized. But this organization introduces an overhead into the memory operation, since mechanisms to bring the required portion of the programs and data into primary memory as needed will have to be devised. These mechanisms form what is called a *virtual memory* scheme.

In a virtual memory scheme, the user assumes that the total memory capacity (primary plus secondary) is available for programming. The oper-

ating system manages the moving in and out of portions (segments or pages) of program and data into and out of the primary memory.

Even with current technologies, the primary memory hardware is slow compared to the processing unit hardware. To reduce this speed gap, a small but faster memory is usually introduced between the main memory and the processing unit. This memory block is called *cache memory* and is usually 10 to 100 times faster than the primary memory. A virtual memory mechanism similar to that between primary and secondary memories is then needed to manage operations between main memory and cache. The set of instructions and data that are immediately needed by the processing unit are brought from the primary memory into cache and retained there. A parallel fetch operation is possible in that while the cache unit is being filled from the main memory, the processing unit can fetch from the cache, thus narrowing the memory-to-processor speed gap.

Note that the registers in the processing unit are temporary storage devices. They are the fastest components of the computer system memory.

Thus, in a general purpose computer system there is a memory hierarchy in which the highest speed memory is closest to the processing unit and is most expensive. The least expensive and slowest memory devices are farthest from the processing unit. Figure 3.8 shows the memory hierarchy.

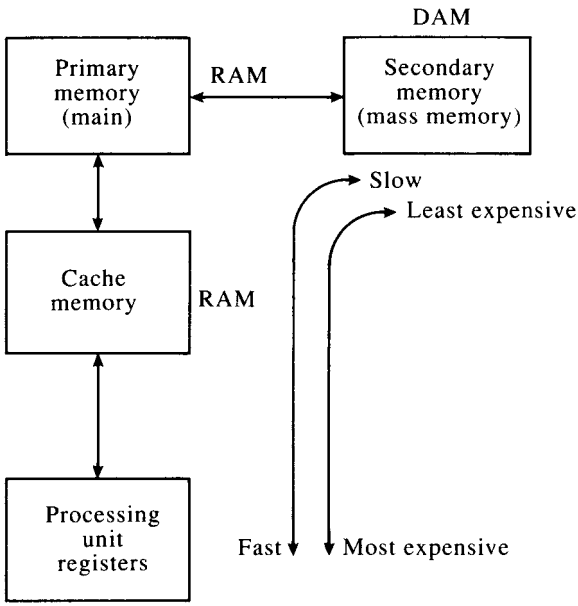



**Figure 3.8** Memory hierarchy

Chapter 8 gives further details on memory hierarchy and virtual memory schemes.

## 3.3 MEMORY DEVICES AND ORGANIZATIONS

The basic property that a memory device should possess is that it must have two well-defined states that can be used for the storage of binary information. In addition, the ability to switch from one state to another (i.e., reading and writing a 0 or 1) is required, and the switching time must be small in order to make the memory system fast. Further, the cost per bit of storage should be as low as possible.

The address decoding mechanism and its implementation distinguish RAM from non-random-access memory. Since RAM needs to be fast, the address decoding is done all electronically, thus involving no physical movement of the storage media. In a non-random-access memory, either the storage medium or the read/write mechanism (transducers) is usually moved until the appropriate address (or data) is found. This sharing of the addressing mechanism makes nonRAM less expensive than RAM, while the mechanical movement makes it slower than RAM in terms of data access times. In addition to the memory device characteristics, decoding of the external address and read/write circuitry affect the speed and cost of the storage system.

Semiconductor and magnetic technologies have been the popular primary memory device technologies. Magnetic core memories were used extensively as primary memories during the 1970s. They are now obsolete, since the semiconductor memories have the advantages of lower cost and higher speed. One advantage of magnetic core memories is that they are *nonvolatile*. That is, the data are retained by the memory even after the power is turned off. Semiconductor memories, on the other hand, are *volatile*. Either a backup power source must be used to retain the memory contents when power is turned off, or the memory contents are dumped to a secondary memory and restored when needed to circumvent the volatility of these memories.

The most popular secondary storage devices have been magnetic tape and disk. Optical disks are now becoming cost effective with the introduction of compact disk ROMs (CDROM), write-once-read-many-times (or read-mostly) (WORM) disks and erasable disks.

In each technology, memory devices can be organized in various configurations with varying cost and speed characteristics. We will now examine representative devices and organizations of semiconductor memory technology.

### 3.3.1 Random–access Memory Devices

Two types of semiconductor RAMs are now available: *static* and *dynamic*. In a static RAM, each memory cell is built out of a flip-flop. Thus, the content of the memory cell (either 1 or 0) remains intact as long as the power is on. Hence, the memory device is *static*. A dynamic memory cell, however, is built out of a capacitor. The charge level of the capacitor determines the 1 or 0 state of the cell. Because the charge decays with time, these memory cells must be *refreshed* (i.e., recharged) every so often to retain the memory content. Dynamic memories require complex refresh circuits, and because of the refresh time needed, they are slower than static memories. But more dynamic memory cells can be fabricated on the same area of silicon than static memory cells can. Thus, when large memories are needed and speed is not a critical design parameter, dynamic memories are used; static memories are used in speed-critical applications.

#### Static RAM

The major components of RAM are the address decoding circuit, the read/write circuit, and the set of memory devices organized into several words. The memory device that can store a bit and has the appropriate hardware to support decoding and read/write operations is called a *memory cell*.

Flip-flops are used in forming static RAM cells. Figure 3.9(a) shows a memory cell built out of a *JK* flip-flop. The flip-flop is not clocked. When the

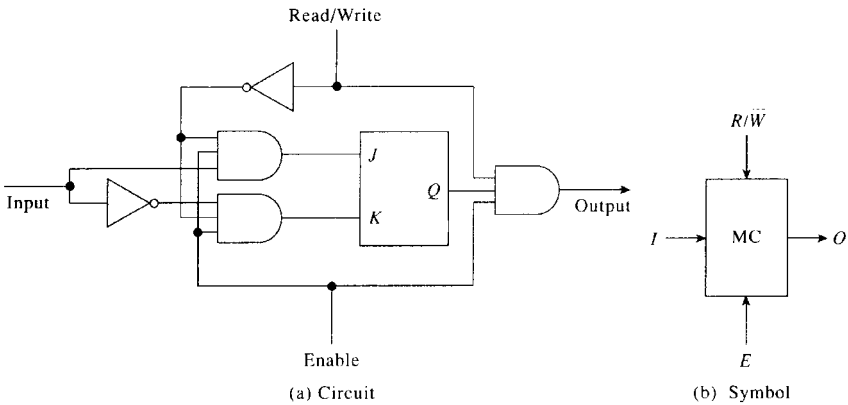



**Figure 3.9** A semiconductor memory cell

enable signal is 1, either the input signal enters the flip-flop or the contents of the flip-flop are seen on the Output based on the value of the read/write signal. If the enable is 0, the cell outputs 0 and also makes $J = K = 0$, leaving the contents of the flip-flop unchanged. The read/write signal is 1 for reading (i.e., output $\leftarrow Q$) and 0 for writing (i.e., $Q \leftarrow$ input). A symbol for this memory cell (MC) is shown in (b).

A $(4 \times 3)$-bit RAM, built out of such memory cells, is shown in Fig. 3.10. The two-bit address in the MAR is decoded by a 2-to-4 decoder, to select one of the four memory words. For the memory to be active, the memory enable line must be 1. If not, none of the words is selected (i.e., all outputs of the decoder are 0). When the memory enable line is 1 and the R/$\overline{\text{W}}$ line is 1, the outputs of MCs enabled by the selected word line will be input into the set of OR gates whose outputs are connected to the output lines. Output lines receive signals from MCs that are in the enabled word only, since all other MC outputs in each bit position are 0. If the memory is enabled and the R/$\overline{\text{W}}$ line is 0, only the selected word will receive the INPUT information.

If the number of words is large, as in any practical semiconductor RAM, the OR gates shown in Fig. 3.10 become impractical. To eliminate these gates, the MCs are fabricated with either open collector or tristate outputs. If open collector outputs are provided, outputs of MCs in each bit position are tied together to form a wired-OR, thus eliminating an OR gate. But the pull-up resistors required and the current dissipation by the gates limit the number of the outputs of gates that can be wire-ORed. MCs with tristate outputs can be used in such limiting cases. Outputs of MCs in each bit position are then tied together to form an output line.

When the number of words in the memory is large, the *linear decoding* technique of Fig. 3.10 results in complex decoding circuitry. In order to reduce the complexity, *coincident decoding* schemes are used. Figure 3.11 shows such a scheme. Here, the address is divided into two parts. $X$ and $Y$. The low-order bits of the address ($Y$) select a column in the MC matrix and the high-order bits of the address ($X$) select a row. The MC selected is the one at the intersection of the selected row and column. When the data word consists of more than one bit, several columns of the selected row are selected by this coincident decoding technique. The enable input of the MC is now obtained by ANDing the $X$ and $Y$ selection lines.

Some commercial memory ICs provide more than one enable signal input on each chip. These multiple enable inputs are useful in building large memory systems employing coincident memory decoding schemes. The outputs of these ICs will also be either open collector or tristate, to enable easier interconnection.

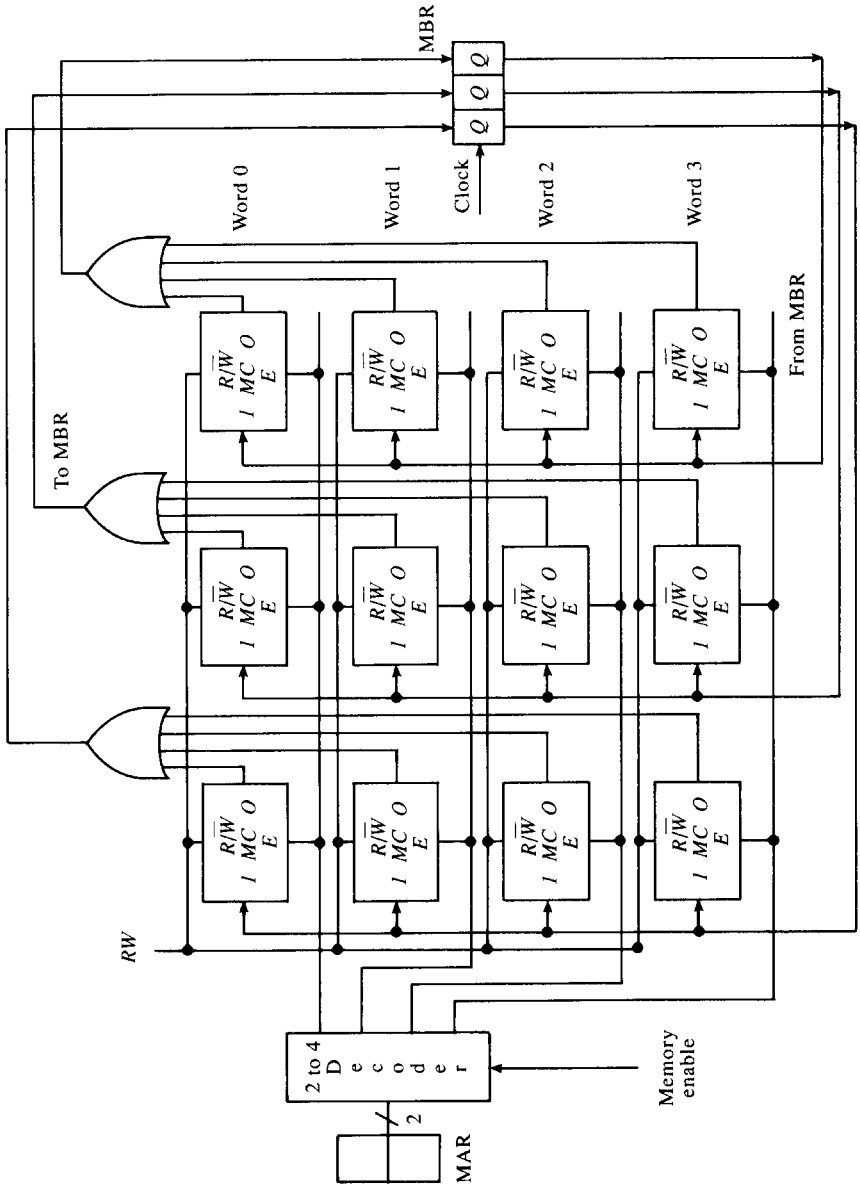



**Figure 3.10** A 4-word, 3-bit semiconductor memory

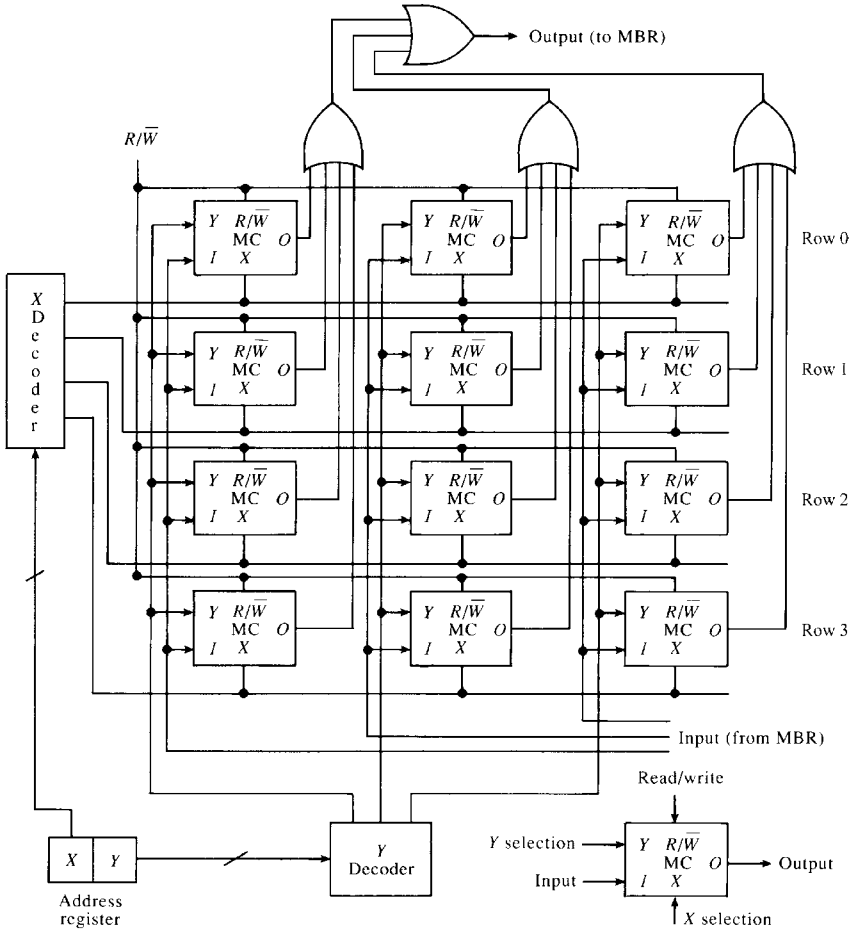



**Figure 3.11** Coincident decoding (1 bit/word; 12 words)

## Dynamic Memory

Several dynamic MC configurations have been used. Figure 3.12 shows the most common dynamic MC (DMC), built from a single MOS transistor and a capacitor. Read and Write control is achieved by using two MOS transistors.

Consider the *n*-channel MOS (NMOS) transistor $Q_1$ in (a). The transistor has three terminals: drain, source, and gate. When the voltage on the gate is positive (and exceeds certain threshold value), the transistor

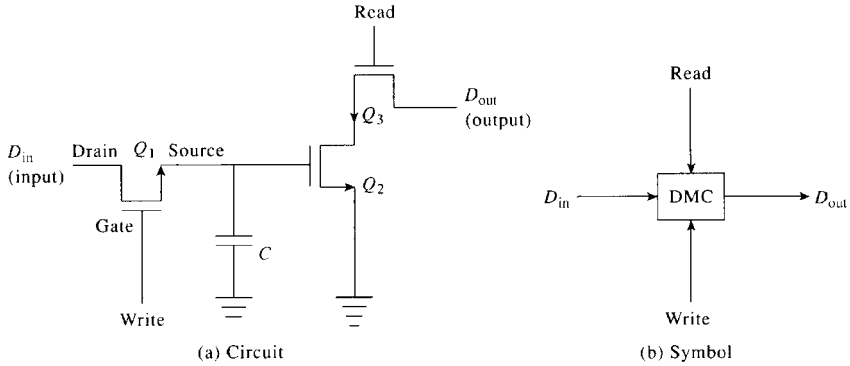


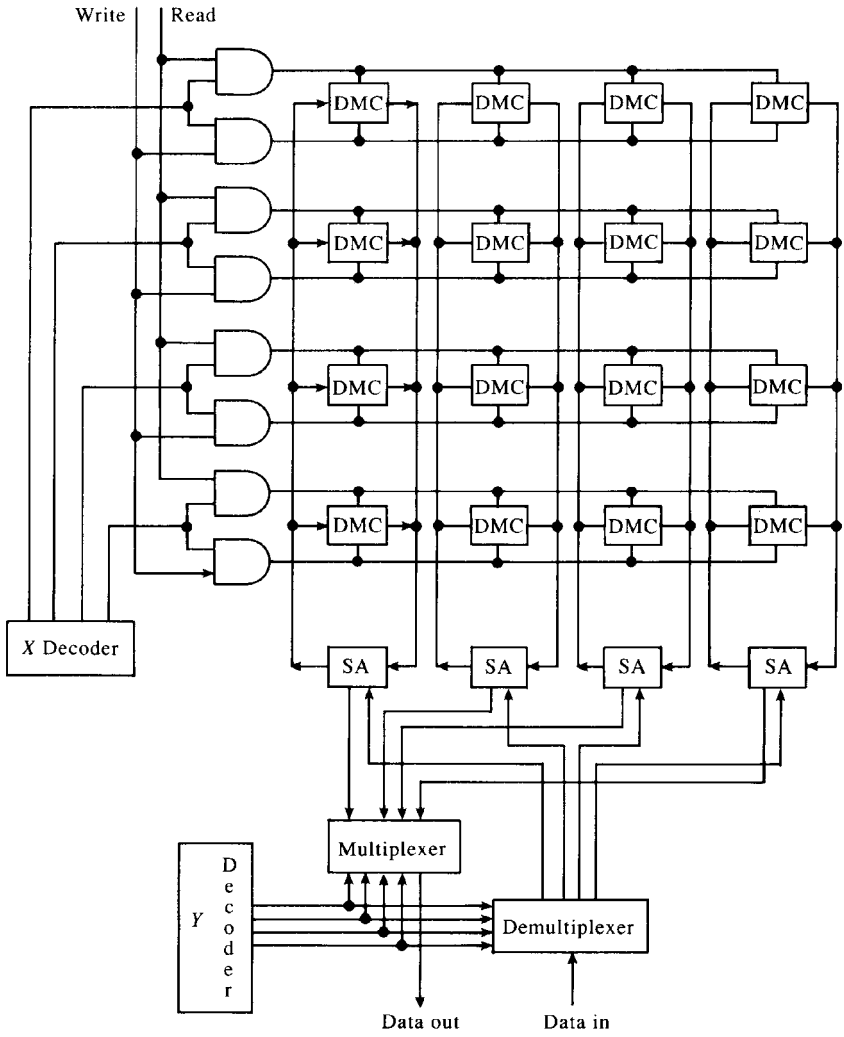

**Figure 3.12** Dynamic memory cell

conducts, thus connecting the drain to the source. If the gate voltage is negative, the transistor is off, thus isolating the drain and the source.

When the Write control is high, $Q_1$ is on and $D_{in}$ is transferred to the gate of $Q_2$, across the capacitor. If $D_{in}$ is high, the capacitor is charged; otherwise, the capacitor is discharged through the gate-to-source resistance of $Q_2$, while the Write is active. A discharged capacitor corresponds to storing a low in the MC and thus a stored low can be maintained indefinitely. A charged capacitor corresponds to storing a high. The high value can be maintained only as long as the capacitor remains charged. In order to fabricate denser memories, the capacitor is made very small and hence the capacitance will be quite small (on the order of a fraction of a picofarad). The capacitor thus discharges in several hundred milliseconds, requiring that the charge be restored, or "refreshed," approximately every two milliseconds.

When the Read control goes high, $Q_3$ is on and the drain of $Q_2$ is connected to $D_{out}$. Since the stored data are impressed on the gate of $Q_2$ and the output is from the drain of $Q_2$, $D_{out}$ will be the complement of the data stored in the cell. The output data are usually inverted by the external circuitry. Note that the read operation is destructive since the capacitor is discharged when the data are read. Thus, the data must be refreshed.

Figure 3.13 shows a $(16 \times 1)$-bit dynamic memory using the DMC of Fig. 3.12. The DMCs are internally organized in a $4 \times 4$ matrix. The high-order two bits of the address select one of the rows. When the read is on, data from the selected row are transferred to sense amplifiers. Since the capacitance of output lines is much higher than the capacitor in the DMC, output voltage is very low; consequently, sense amplifiers are required to detect the data value in the presence of noise. These amplifiers are also used to refresh the memory. The low-order two bits of the address




Note: SA = sense amplifiers
DMC = dynamic memory cell

**Figure 3.13** (16 × 1)-bit dynamic memory

are used to select one of the four sense amplifiers for the one-bit data output. The data in the sense amplifiers are rewritten into the row of DMCs. In order to write a one-bit data, the selected row is first read, and the data in the selected sense amplifier are changed to the new data value just before the rewrite operation.

The need for the refresh results in the requirement of complex refresh circuitry and also reduces the speed of dynamic memory devices. But because of the small size of DMCs, it is possible to fabricate very dense memories on a chip. Usually the refresh operation is made transparent by performing a refresh when the memory is not being otherwise used by the system, thereby gaining some speed. When the memory capacity required is small, the refresh circuitry can be built on the memory chip itself. Such memory ICs are called *integrated dynamic RAMs* (iRAMs). Dynamic memory controller ICs that handle refresh and generate all the control signals for dynamic memory are available. These are used in building large dynamic memory systems.

## Read-only Memory

A read-only memory (ROM) is a random-access memory with data permanently stored in it. When the *n*-bit address is input to the ROM, the data stored at the addressed location are output on its output lines. ROM is basically a combinational logic device. Figure 3.14 shows a four-word ROM with 3 bits per word. The links at junctions of word lines and bit lines are either open or closed depending on whether a 0 or a 1 is stored at the junction, respectively. When a word is selected by the address decoder, each output line (i.e., bit line) with a closed link at its junction with the selected word line will contain a 1 while the other lines contain a 0. In Fig. 3.14, contents of locations 0 through 3 are 101, 010, 111, and 001, respectively.

Two types of ROMs are commercially available, *mask-programmed* ROMs and *user-programmed* ROMs. Mask-programmed ROMs are used when a large number of ROM units containing a particular program and/or

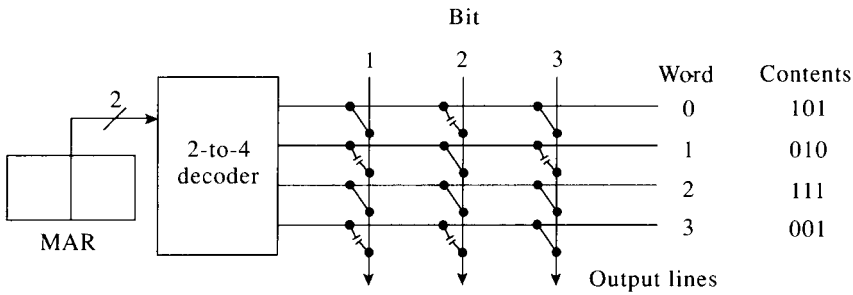



**Figure 3.14** A 4-word, 3-bit-per-word ROM

data is required. The IC manufacturer can be asked to "burn" the program and data into the ROM unit. The program is given by the user and the IC manufacturer prepares a mask and uses it to fabricate the program and data into the ROM as the last step in the fabrication. The ROM is thus custom fabricated to suit the particular application. Since custom manufacturing of an IC is expensive, mask-programmed ROMs are not cost effective unless the application requires a large number of units, thus spreading the cost among the units. Further, since the contents of these ROMs are unalterable, any change requires new fabrication.

A user-programmable ROM (programmable ROM or PROM) is fabricated with either all 1s or all 0s stored in it. A special device called a *PROM programmer* is used by the user to "burn" the required program, by sending the proper current through each link. Contents of this type of ROM cannot be altered after initial programming. Erasable PROMs (EPROM) are available. An ultraviolet light is used to restore the content of an EPROM to its initial value of either all 0s or all 1s. It can then be reprogrammed using a PROM programmer. Electrically alterable ROMs (EAROMs) are another kind of ROM that uses a specially designed electrical signal to alter its contents.

ROMs are used for storing programs and data that are not expected to change during program execution (i.e., in real time). They are also used in implementing complex Boolean functions, code converters, and the like. An example of ROM-based implementation follows.

---

**Example 3.1** Implement a binary coded decimal (BCD)-to-Excess-3 decoder using a ROM.

Figure 3.15 shows the BCD-to-Excess-3 conversion. Since there are ten input (BCD) combinations, a ROM with sixteen words ($2^3 < 10 < 2^4$) must be used. The first ten words of the ROM will contain the ten Excess-3 code words. Each word is 4 bits long. The BCD input appears on the four address input lines of the ROM. The content of the addressed word is output on the output lines. This output is the required Excess-3 code.

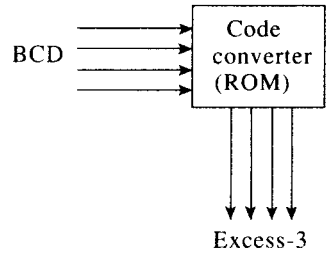
---

## Programmable Logic Arrays (PLA)

In a ROM with $n$ address lines, there will be $2^n$ words. In some applications, it is not necessary to have the complete $2^n$ word space. PLAs may be used instead of ROMs in such applications. We will illustrate the utility of PLAs with the following example.

| BCD | ROM address | ROM content (Excess-3) |
|---|---|---|
| 0000 | 0 | 0011 |
| 0001 | 1 | 0100 |
| 0010 | 2 | 0101 |
| 0011 | 3 | 0110 |
| 0100 | 4 | 0111 |
| 0101 | 5 | 1000 |
| 0110 | 6 | 1001 |
| 0111 | 7 | 1010 |
| 1000 | 8 | 1011 |
| 1001 | 9 | 1100 |

(a) Code conversion table




(b) Code converter

**Figure 3.15** ROM-based implementation of code converter

---

**Example 3.2** Consider the implementation of a four-variable Boolean function:

$$F(A, B, C, D) = AB'C + C'D'.$$

A ROM can be used to implement this function. The four address lines correspond to variables $A, B, C$, and $D$. The sixteen 1-bit words of the ROM will each contain the value of $F$, corresponding to each input combination. When a particular combination of inputs is provided to the ROM (as an address), the corresponding $F$ value is output by the ROM.

Figure 3.16 shows a PLA implementation of the above function. A PLA consists of an AND array and an OR array. These semiconductor arrays are fabricated to provide AND and OR functions at the intersections of the horizontal and vertical lines (in practice, they are NAND-NAND or NOR-NOR arrays). A vertical line corresponds to either an input variable or its complement. Each horizontal line in the AND array implements a product term. Appropriate junctions of the AND array are connected to form product terms. This can be done during IC fabrication. Each vertical line in the OR array forms the OR of the connected product terms. There will thus be a vertical line in OR array corresponding to each output. In Fig. 3.16, two horizontal lines are used to implement the two product terms in $F$. These lines are ORed to form $F$.

---

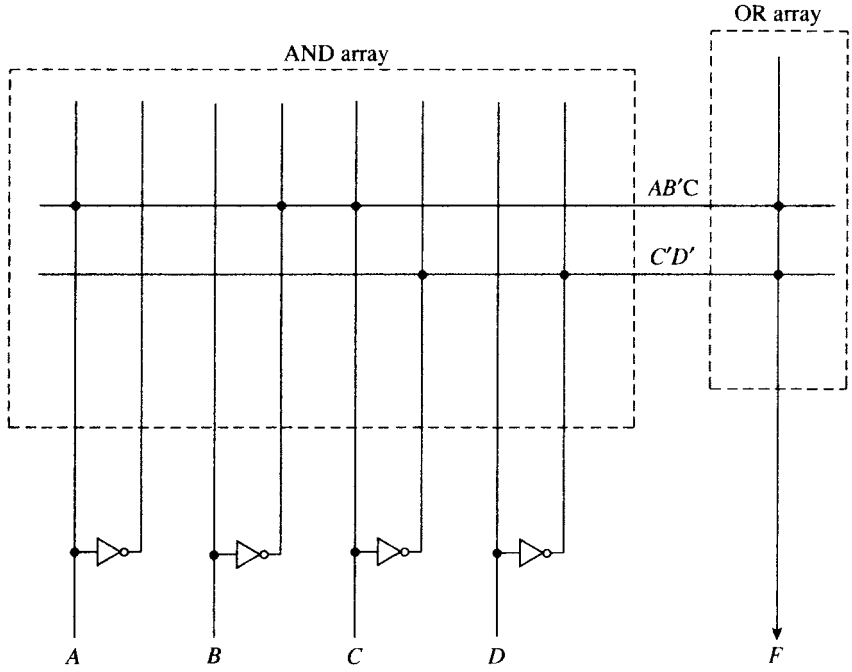



**Figure 3.16** PLA implementation of $AB'C + C'D'$

Just as in the case of ROM, the user provides a PLA program to the IC manufacturer, who programs the PLA during fabrication. Thus, PLAs can be used as ROMs. To implement an $n$ variable function, a ROM with $2^n$ words is needed. Since all the $2^n$ product terms do not occur in practical functions, a PLA can be more cost effective because the PLA uses one horizontal line in the AND array for each product term in the function. For example, the four-variable function of Example 3.2 requires only two horizontal lines in the PLA implementation. It would have required a $16 \times 1$ ROM.

Standard PLA pattern of various configurations are available from IC manufacturers. These configurations differ with respect to the number of AND and OR lines, the number of inputs, and the number of outputs. Both mask-programmed and user-programmable PLAs are available.

PLAs have traditionally being combinational logic devices that can be programmed to implement logic functions. Implementation of sequential circuits required and addition of flip-flops external to the PLA. Programmable logic sequencers (PLS), which are PLAs with flip-flops, are now available.

**Example 3.3** Figure 3.17 shows the implementation of the 1011 sequence detector of Example 2.4, using a PLA and two flip-flops. The next-state and output circuits are realized by the PLA.

### 3.3.2 Associative Memory

A typical associative memory cell (AMC) built from a *JK* flip-flop is shown in Fig. 3.18. The response is 1 when either the data bit ($D$) and the memory bit ($Q$) match while the mask bit ($M$) is 1, or when the mask bit is 0 (corresponding to a "do not compare"). A truth table and a block diagram for the simplified cell are also shown in the figure. In addition to the response circuitry, an AMC will have read, write, and enable circuits similar to the RAM cell shown in Fig. 3.9.

A 4-word, 3-bits-per-word AM built out of the above cells is shown in Fig. 3.19. The data and mask registers are each three bits long. The word select register of Fig. 3.3 is neglected here. Hence, all memory words are

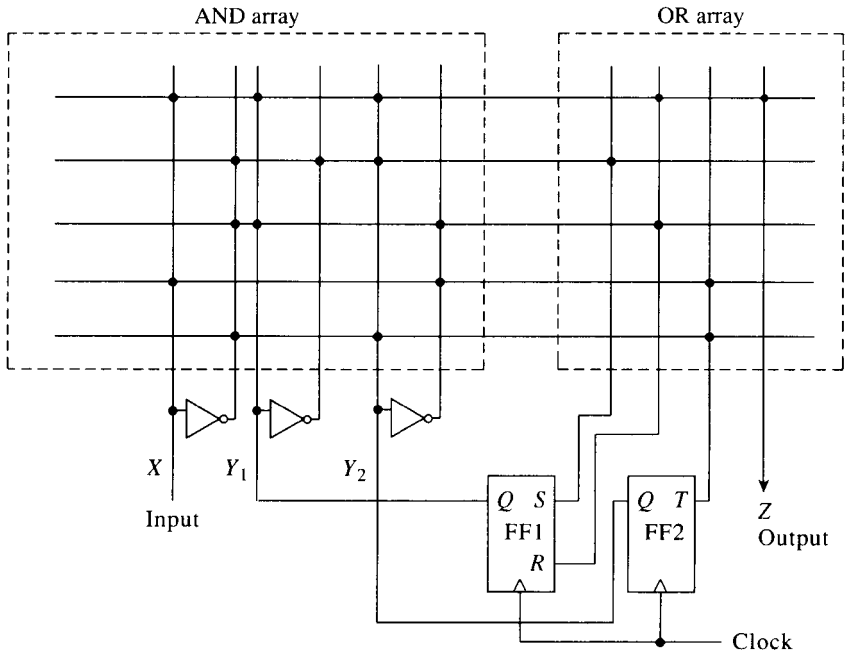



**Figure 3.17** Implementation of 1011 sequence detector (see Example 2.4)

| Data ($D$) | Mask($M$) | $Q$ | Response |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

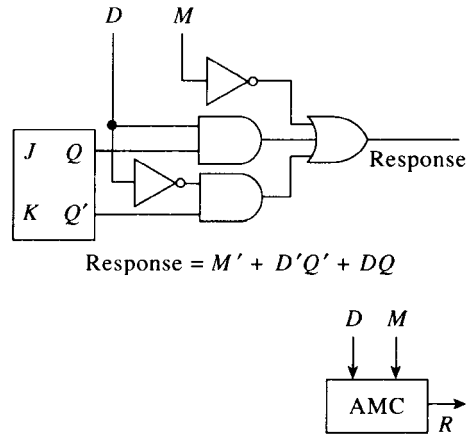



**Figure 3.18** A simplified AMC

selected for comparison with the data register. Response outputs of all cells in a word are ANDed together to form the word response signal. Thus, a word is a respondent if the response output of each and every cell in the word is a 1. The MMR circuit shown selects the first respondent. The first respondent then drives to 0 the response outputs of other words following it.

Input (or write) and output (or read) circuitry is also needed. It is similar to that of the RWM system shown in Fig. 3.10, and hence is not shown in Fig. 3.19.

Small AMs are available as IC chips. Their capacity is on the order of eight bits per word by eight words. Larger AM systems are designed using RAM chips. As such, these memories can be used in both RAM and CAM modes.

Because of their increased logic complexity, CAM systems cost much more than RWMs of equal capacity.

### 3.3.3 Sequential-access Memory Devices

Magnetic tape is the most common sequential-access memory device. A magnetic tape is a mylar tape coated with magnetic material (similar to that used in home music systems) on which data is recorded as magnetic patterns. The tape moves past a read/write head to read or write data.

Figure 3.20 shows the two popular tape formats: the reel-to-reel tape used for storing large volumes of data (usually with large-scale and mini-computer systems) and the cassette tape used for small data volumes

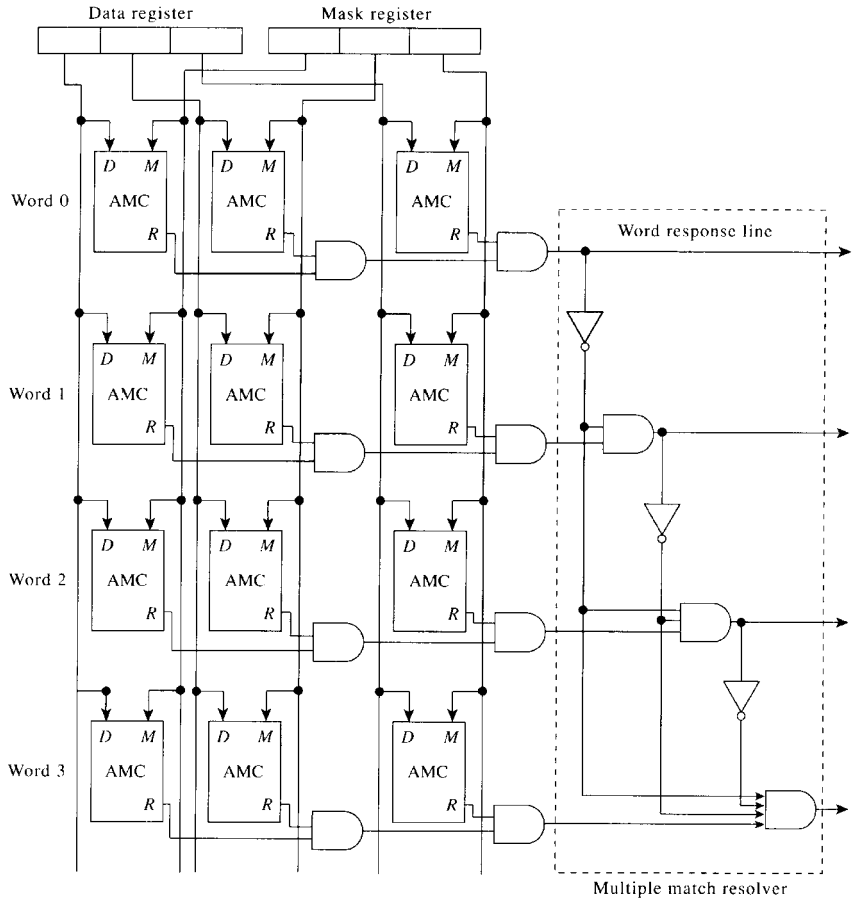



**Figure 3.19** A 3-bit, 4-word AM

(usually with microcomputer systems). Data are recorded on tracks. A track on a magnetic tape runs along the length of the tape and occupies a width just sufficient to store a bit. On a nine-track tape, for example, the width of the tape is divided into nine tracks and each character of data is represented with 9 bits (one bit on each track). One or more of these bits is usually a parity bit, which facilitates error detection and correction.

Several characters grouped together form a *record*. The records are separated by an inter-record gap (about $\frac{3}{4}$ inch and an end-of-record mark (which is a special character). A set of records forms a *file*. The files are separated by an end-of-file mark and a gap (about 3 inches). On cassette

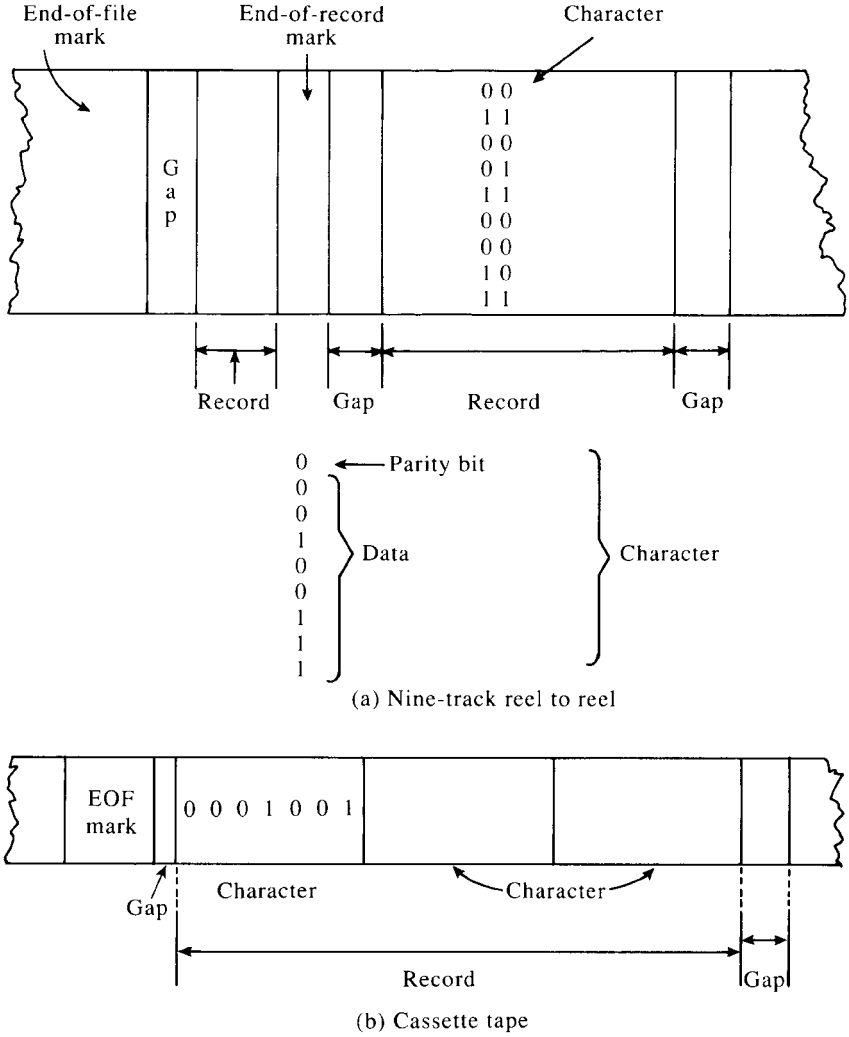



**Figure 3.20** Magnetic tape formats

tapes, the data are recorded in a serial mode on one track as shown in Fig. 3.20(b).

Recording, or *writing*, on magnetic devices is the process of creating magnetic flux patterns on the device; the sensing of the flux pattern when the medium moves past the read/write head constitutes the *reading* of the data. In reel-to-reel tapes, the data are recorded along the tracks digitally. In a

cassette tape, each bit is converted into an audio frequency and is recorded. Digital cassette recording techniques are also becoming popular. Note that the information on a magnetic tape is nonvolatile.

Magnetic tapes permit recording of vast amounts of data at a very low cost. But the access time, being a function of the position of the data on the tape with respect to the read/write head position along the length of the tape, can be very long. Sequential-access devices thus form low-cost secondary memory devices, which are primarily used for storing the data that does not have to be used frequently, such as system backup, archival storage, transporting data to other sites, etc.

### 3.3.4 Direct-access Storage Devices

Magnetic and optical disks are the most popular direct- or semi-random-access storage devices. Accessing data in these devices requires two steps: random or direct movement of read/write heads to the vicinity of data, followed by a sequential access. These mass-memory devices are used as secondary storage devices in a computer system for storing data and programs.

#### Magnetic Disk

A magnetic disk (see Fig. 3.7) is a flat circular surface coated with a magnetic material, much like a phonograph record. Several such disks are mounted on a rotating spindle. Each surface will have a read/write head. Each surface is divided into several concentric circles (tracks). By first positioning read/write heads to the proper track, the data on the track can be accessed sequentially. A track is normally divided into several sectors, each of which correspond to a data word. The address of a data word on the disk thus corresponds to a track number and a sector number.

Magnetic disks are available as either *hard disks* or *floppy disks*. The data storage and access formats for both types of disks are the same. Floppy disks have a flexible disk surface and are very popular storage devices, especially for microcomputer systems, although their data transfer rate is slower than that of hard disk devices.

#### Optical Disk

Three types of optical disks are available. *Compact disk ROMs* (CDROM) are similar to mask-programmed ROMs in which the data is stored on the

disk during the last stage of disk fabrication. The data, once stored, cannot be altered. *Write-once-read-many-times* (or read-mostly) (WORM) optical disks allow writing the data once. The portions of the disk that are written once cannot be altered. *Erasable* optical disks are similar to magnetic disks that allow repeated erasing and storing of data.

Manufacturing of CDROMs is similar to that of phonograph records except that digital data is recorded by burning pit and no-pit patterns on the plastic substrate on the disk with a laser beam. The substrate is the metallized and sealed. While reading the recorded data, 1s and 0s are distinguished by the differing reflectivity of an incident laser beam. Since the data once recorded cannot be altered, the application for CDROMs has been in storing the data bases that do not change. The advantage of CDROMs over magnetic disks is their low cost, high density, and nonerasability.

Erasable optical disks use a coating of a magneto-optic material on the disk surface. In order to record data on these disks, a laser beam is used to heat the magneto-optic material in the presence of a bias field applied by a bias coil. The bit positions that are heated take on the magnetic polarization of the bias field. This polarization is retained when the surface is cooled. If the bias field is reversed and the surface is heated, the data at the corresponding bit position is erased. Thus, changing the data on the disk requires a two-step operation. All the bits in a track are first erased and new data is then written onto the track. During reading, the polarization of the read laser beam is rotated by the magnetic field. Thus the polarization of the laser beam striking the written bit positions is different from that of the rest of the media.

WORM devices are similar to erasable disks except that the portions of the disk that are written once cannot be erased and rewritten.

Optical disk technology offers densities of about 50,000 bits and 20,000 tracks per inch, resulting in a capacity of about 600 megabytes per $3\frac{1}{2}$-inch disk. Corresponding numbers for the magnetic disk technology are 150,000 bits and 2000 tracks per inch, or 200 megabytes per $3\frac{1}{2}$-inch disk. Thus optical storage offers a 3-to-1 advantage over magnetic storage in terms of capacity. It also offers a better per-unit storage cost. The storage densities of magnetic disks are also rapidly increasing, especially since the advent of vertical recording formats.

The data transfer rates of optical disks are much lower than magnetic disks due to the following factors: It takes two revolutions to alter the data on the track; the rotation speed needs to be about half that of magnetic disks to allow time for heating and changing of bit positions; and, since the optical read/write heads are bulkier than their magnetic counterparts, the seek times are higher.

The storage density and the speed of access of disks improve so rapidly that a listing of such characteristics soon becomes outdated. Refer to the magazines listed in the reference section of this chapter for such details.

## 3.4 MEMORY SYSTEM DESIGN USING ICs

Refer to Appendix C for the details of representative memory ICs. Memory system designers use such commercially available memory ICs to design memory systems of required size and other characteristics. The major steps in such memory designs are the following:

1. Based on speed and cost parameters, determining the type of memory ICs (static or dynamic) to be used in the design.
2. Selecting an available IC of the type selected above, based on access time requirements and other physical parameters, such as the restriction on the number of chips that can be used and the power requirements. It is generally better to select an IC with the largest capacity in order to reduce the number of ICs in the system.
3. Determining the number of ICs needed $N =$ (total memory capacity)/(chip capacity).
4. Arranging the above $N$ ICs in a $P \times Q$ matrix, where $Q =$ (number of bits per word in memory system)/(number of bits per word in the IC) and $P = N/Q$.
5. Designing the decoding circuitry to select a unique word corresponding to each address.

We have not addressed the issue of memory control in this design procedure. The control unit of the computer system, of which the memory is a part, should produce control signals to strobe the address into the MAR, enable read/write, and gate the data in and out of MBR at appropriate times.

The following example illustrates the design.

---

**Example 3.4** Design a 4K $\times$ 8 memory, using Intel 2114 RAM chips.

1. Number of chips needed $= \dfrac{\text{total memory capacity}}{\text{chip capacity}} = \dfrac{4\text{K} \times 8}{1\text{K} \times 4} = 8$
2. The memory system MAR will have 12 bits, since 4K $= 4 \times 1024 = 2^{12}$; the MBR will have 8 bits.

3. Since 2114s are organized with four bits per word, two chips are used in forming a memory word of eight bits. Thus, the eight 2114s are arranged in four rows, with two chips per row.
4. The 2114 has 10 address lines. The least significant 10 bits of the memory system MAR are connected to the 10 address lines of each 2114. A 2-to-4 decoder is used to decode the most significant two bits of the MAR, to select one of the four rows of 2114 chips through the $\overline{CS}$ signal on each 2114 chip.
5. I/O lines of chips in each row are connected to the MBR. Note that these I/O lines are configured as tristate. The $\overline{WE}$ lines of all the 2114 chips are tied together to form the system $\overline{WE}$.

The memory system is shown in Fig. 3.21. Note that the number of bits in the memory word can be increased in multiples of 4 simply by including additional columns of chips. If the number of words needs to be extended beyond 4K, additional decoding circuitry will be needed.

---

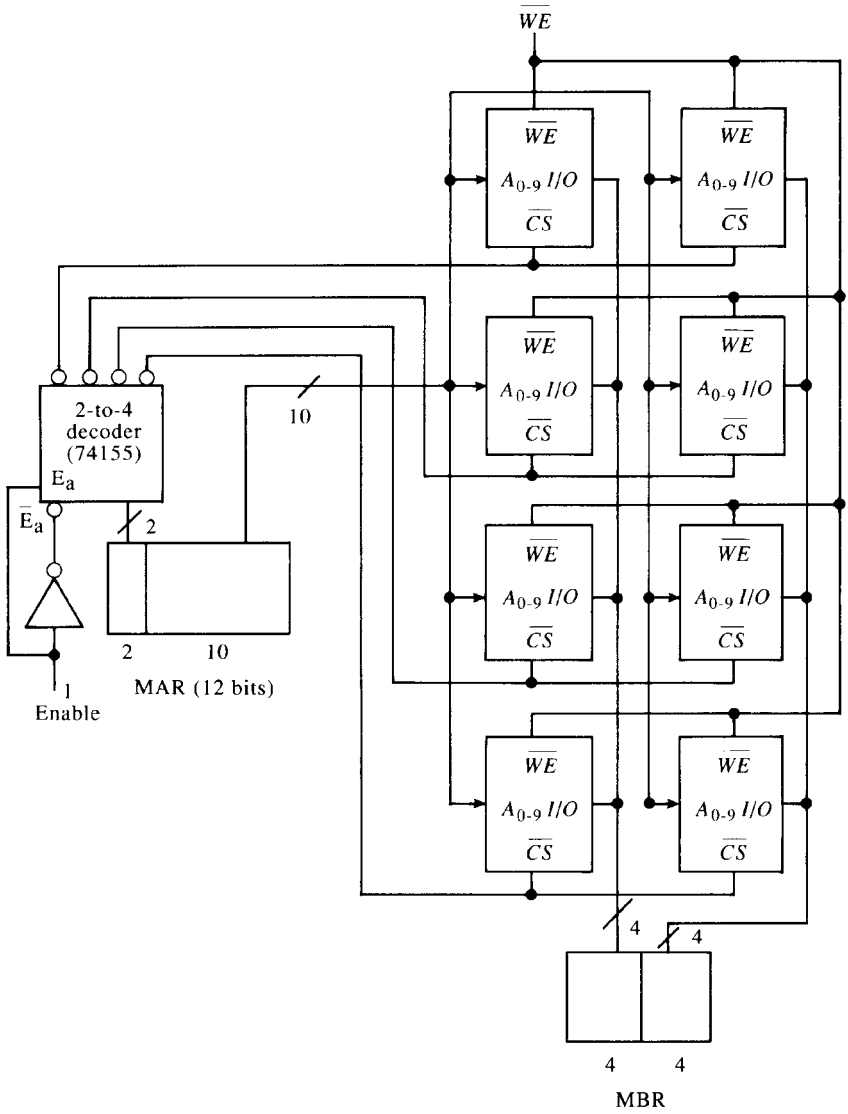
**Example 3.5** Figure 3.22 shows a microcomputer system using the Intel 8088 processor. The memory subsystem is built out of Intel 2118 64K $\times 1$ dynamic RAM ICs, which are equivalent to TMS4116. There are four memory banks, each containing 64K eight-bit words. The 8202A receives the addresses and read/write signals from the processor to control the memory read/write operations.

---

## 3.5 SUMMARY

The four basic types of memories (random-access, content-addressable, sequential-access, and semi-random-access) were introduced in this chapter. Design of typical memory cells and large memory systems with representative ICs were described. The main memory of any computer system is now built out of semiconductor RAM devices, since advances in semiconductor technology have made them cost effective. Magnetic disks are the current popular secondary memory devices, both in hard and floppy formats. Optical memories that use holographic techniques to store and retrieve data have emerged from the laboratory, promising faster, denser memories.

Speed-versus-cost tradeoff is the basic parameter in designing memory systems. Although large-capacity semiconductor RAMs are now available and used in system design, memory hierarchies consisting of a




Note: Power and ground not shown

**Figure 3.21** A 4K × 8 memory

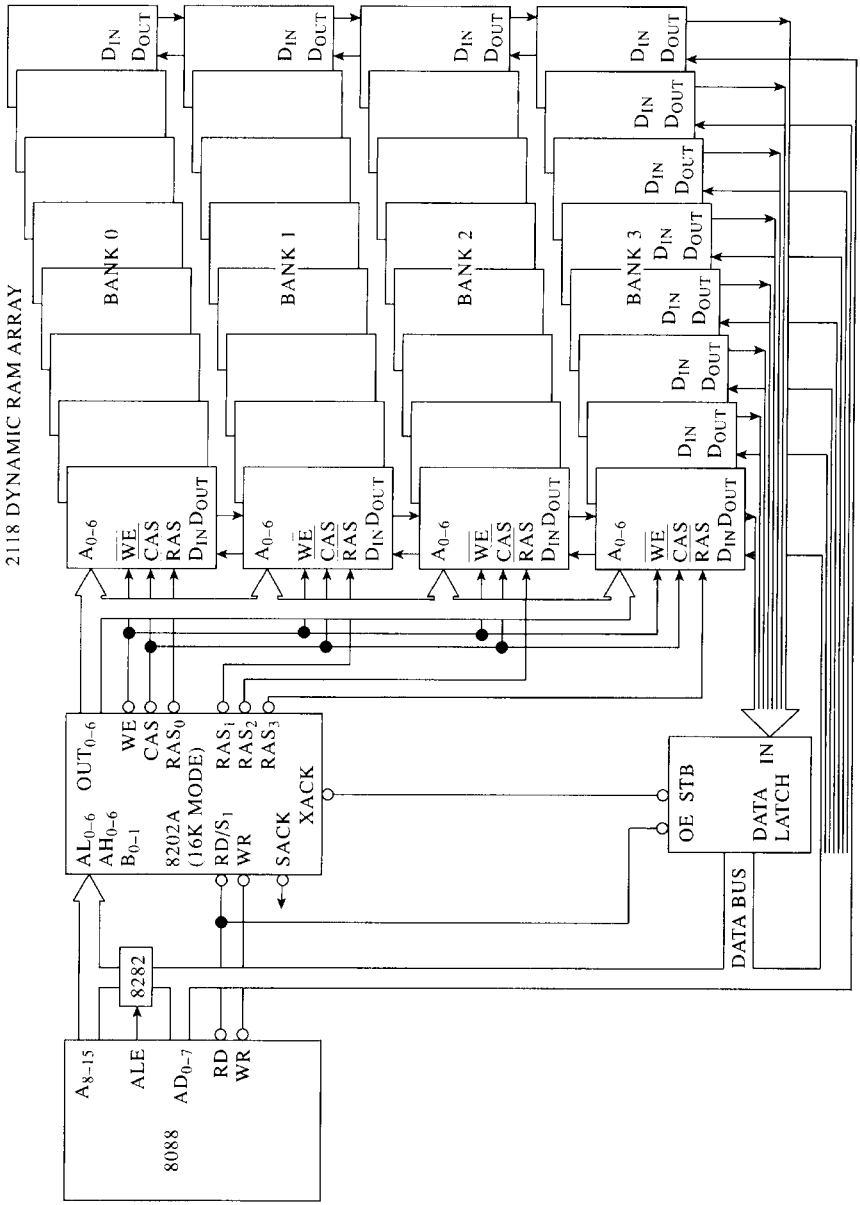



**Figure 3.22** A microcomputer system based on the Intel 8088 processor. (Reprinted by permission of Intel Corp. Copyright 1983. All mnemonics copyright Intel Corp. 1986.)

combination of random- and direct-access devices are still seen even on smaller computer systems. Chapter 8 provides further details on the design of such hierarchies.

The speed and cost characteristics of memory devices keep improving almost daily. As a result, we have avoided listing those characteristics of the devices currently available. Refer to the manufacturers' manuals and magazines listed in the references section of this chapter for such information.

## REFERENCES

Bartee, T. C. *Computer Architecture and Logic Design*, New York, NY: McGraw-Hill, 1991.

*Bipolar memory Data Manual*, Sunnyvale, CA: Signetics, 1987.

*Bipolar Microcomputer Components Data Book*, Dallas, TX: Texas Instruments, 1987.

*Computer*, Los Alamitos, CA: IEEE Computer Society, Published Monthly.

Iyer, S. S. and Kalter, H. L. ''Embedded DRAM Technology: opportunities and challenges'', *IEEE Spectrum*, April 1999, vol. 36, No. 4, pp. 56–64.

*Electronic Design*, Hasbrouck Heights, NJ: Penton Publishing, Published twice monthly.

*Computer Design*, Northbrook, IL: PennWell Publishing, Published monthly.

Shiva, S. G. *Introduction to Logic Design*, New York, NY: Marcel Dekker, 1998.

## PROBLEMS

3.1 Complete the following table:

| Memory-system capacity | Number of bits in MAR | Number of bits in MBR | Number of chips needed if chip capacity is | | |
|---|---|---|---|---|---|
| | | | 1K × 4 | 2K × 1 | 1K × 8 |
| 64K × 4 | | | | | |
| 64K × 8 | | | | | |
| 32K × 4 | | | | | |
| 32K × 16 | | | | | |
| 32K × 32 | | | | | |
| 10K × 8 | | | | | |
| 10K × 10 | | | | | |

3.2 What is the storage capacity and maximum data transfer rate of
   a. a magnetic tape, 800 bits per inch, 2400 feet long, 10 inches per second?
   b. A magnetic disk with 50 tracks and 4K bits per track, rotating at 3600 r.p.m.?
   What is the maximum access time assuming a track-to-track move time of 1 millisecond?

3.3 Implement the following functions using a PLA:

$$F_1(A, B, C) = A'BC + AB' + ABC'$$
$$F_2(A, B, C) = A'B' + A'BC + C'$$

Is it advantageous to reduce the Boolean functions before implementing them using PLAs?

3.4 Implement the sequential circuit of Problem 2.7(a) using PLA.

3.5 Redesign the memory cell of Fig. 3.9 to make it suitable for coincident decoding (i.e., two enable signals).

3.6 Design the decoding circuitry for a 2K $\times$ 4 memory IC, using linear and coincident decoding schemes. Make the memory cell array as square as possible in the latter case. Compare the complexities of these designs.

3.7 A 256-word memory has its words numbered from 0 to 255. Define the address bits for each of the following. Each address bit should be specified as 0, 1, or *d* (don't-care).
   a. Word 48.
   b. Lower half of the memory (words 0 through 127).
   c. Upper half of the memory (words 128 through 255).
   d. Even memory words (e.g., 0, 2, 4).
   e. Any of the eight words 48 through 55.

3.8 Design a 16K $\times$ 8 memory using the following ICs:
   a. 1024 $\times$1 b. 2K $\times$ 4 c. 1K $\times$ 8
   Each IC has on-chip decoding, tristate outputs, and an enable pin.

3.9 In each of the designs of Problem 3.6, identify the exact physical location (i.e., the chip(s) and the word(s) within the chips) where the following memory addresses lie:
   a. Word 20
   b. Word 1028
   c. Word 4096

3.10 Arrange the 16 chips needed in the design of Problem 3.8(b) as a 4 $\times$ 4 array and design the decoding circuitry.

3.11 You are given a 16K $\times$ 32 RAM unit. Convert it into a 64K $\times$ 8 RAM. Treat the 16K $\times$ 32 RAM as one unit that you cannot alter. Only include logic external to the RAM unit.

3.12 A processor has a memory addressing range of 64K with eight bits per word. The lower and upper 4K of the memory must be ROM, and the rest of the memory must be RAM. Use 1K $\times$ 8 ROM and 4K $\times$ 4 RAM chips and design the memory system. Design a circuit to generate an error signal if an attempt is made to write into a ROM area.

3.13 An 8K memory is divided into 32 equal-size blocks (or pages) of 256 words each. The address bits are then grouped into two fields: the page number and the number of the memory word within a page. Draw the memory, MAR, and MBR configurations.

3.14 Four of the 32 pages of the memory of Problem 3.13 must be accessible at any time. Four auxiliary five-bit registers, each containing a page address, are used for this purpose. The processor outputs a 10-bit address, the most significant two bits of which select one of the auxiliary registers and the least significant eight bits select a word within a page, the page number obtained from the selected auxiliary register. Design the circuit to convert the 10-bit address output by the processor into the 13-bit address required.

3.15 Assume that a dynamic RAM controller is available for an 8K RAM with multiplexed addresses. Draw the schematic diagram of the controller showing the address (multiplexed) and the data and control signals. Describe its operation.

3.16 The computations in a particular computer system with an associative memory require that the contents of any field in the associative memory would be incremented by 1.

a. Design a circuit to perform this increment operation on a selected field in all memory words that respond to a search.
b. Extend the circuit in (a) to accommodate the simultaneous addition of a constant value greater than 1 to the selected field of all respondents.

3.17 Using appropriate shift registers, draw the schematic for a FIFO memory (queue) with eight words and four bits per word. It is required that the memory provide the queue full and queue empty conditions as the status signals. Derive the conditions under which these status signals are valid. Design the circuit.

3.18 Develop a circuit similar to the one in the above problem for an 8 $\times$ 4 LIFO (stack) memory.

3.19 In practice, a RAM is used to simulate the operation of a stack by manipulating the address value in the MAR. Here, the data values in the stack do not actually shift during PUSH and POP operations. Rather, the address of the RAM location corresponding to the data input and output is changed appropriately. Determine the sequence of operations needed for PUSH and POP.

3.20 Develop the schematic for a two-port memory. Each port in the memory corresponds to an MAR and an MBR. Data can be simultaneously accessed by these ports. In other words:
   a. Data can be read from both ports simultaneously.
   b. Data can be written into two different locations simultaneously.
   c. If both ports address the same location during a write operation, the data from PORT 1 should be written. That is, PORT 1 has higher priority.

# 4

# A Simple Computer: Organization and Programming

Chapters 1, 2, and 3 provided the hardware design and analysis information needed to understand the organization and design of a computer. The purpose of this chapter is to introduce the terminology and basic functions of a simple but complete computer, mainly from a programmer's (user's) point of view. We will call the simple hypothetic computer ASC (A Simple Computer). Although ASC appears very primitive in comparison with any commercially available machine, its organization reflects the basic structure of the most complex modern computer. The instruction set is limited but complete enough to write powerful programs. Assembly language programming and understanding of assembly process are a must for a system designer. We will not outline the tradeoffs involved in selecting the architectural features of a machine in this chapter. Subsequent chapters of this book, however, deal with such tradeoffs. The detailed hardware design of ASC is provided in Chapter 5. Chapters 6 through 11 examine selected architectural attributes of commercially available computer systems.

## 4.1 A SIMPLE COMPUTER

We will assume that ASC is a 16-bit machine; hence the unit of data manipulated by and transferred between various registers of the machine is 16 bits long. It is a binary, stored-program computer and uses 2s complement representation for negative numbers. Since one can address 64K-memory words with a 16-bit address, we will assume a memory with 64K 16-bit words. A 16-bit-long memory address register (MAR) is thus required. The memory buffer register (MBR) is also 16 bits long. MAR stores the address of a memory location to be accessed, and MBR receives the data from the

memory word during a memory-read operation and retains the data to be written into a memory word during a memory-write operation. These two registers are not normally accessible by the programmer.

In a stored-program machine, programs are stored in the memory. During the execution of the program, each instruction from the stored program is first *fetched* from the memory into the control unit and then the operations called for by the instruction are performed (i.e., the instruction is *executed*). Two special registers are used for performing fetch-execute operations: A *program counter* (PC) and an *instruction register* (IR). The PC contains the address of the instruction to be fetched from the memory and is usually incremented by the control unit to point to the next instruction address at the end of an instruction fetch. The instruction is fetched into IR. The circuitry connected to IR decodes the instruction and generates appropriate control signals to perform the operations called for by the instruction. PC and IR are both 16 bits long in ASC.

There is a 16-bit *accumulator register* (ACC) used in all arithmetic and logic operations. As the name implies, it accumulates the result of arithmetic and logic operations.

There are three *index register* (INDEX 1, 2, and 3) used in manipulation of addresses. We will discuss the function of these registers later in this chapter.

There is a 5-bit *processor status register* (PSR) whose bits represent carry (C), negative (N), zero (Z), overflow (V) and interrupt enable (I) conditions. If an operation in the arithmetic/logic unit results in a carry from the most significant bit of the accumulator, then the carry bit is set. Negative and zero bits indicate the status of the accumulator after each operation that involves the accumulator. The interrupt-enable flag indicates that the processor can accept an interrupt. Interrupts are discussed in Chapter 6. The overflow bit is provided to complete the PSR illustration but is not used further in this chapter. A *console* is needed to permit operator interaction with the machine. ASC console permits the operator to examine and change the contents of memory locations and initialize the program counter. Power ON/OFF and START/STOP controls are also on the console. The console has a set of sixteen switches through which a 16-bit data word can be entered into ASC memory. There are sixteen lights (monitors) that can display 16-bit data from either a memory location or a specified register. To execute a program, the operator first loads the programs and data into the memory, then sets the PC contents to the address of the first instruction in the program and STARTs the machine. The concept of a console is probably old-fashioned, since most of the modern machines are designed to use one of the input/output devices (such as a terminal) as the system console. A

console is necessary during the debugging phase of the computer design process. We have included a console to simplify the discussion of program loading and execution concepts.

During the execution of the program, additional data input (or output) is done through an input (or output) device. For simplicity we assume that there is one input device that can transfer a 16-bit data word into the ACC and one output device that can transfer the 16-bit content of the ACC to an output medium. It could very well be that the keyboard of a terminal is the input device and its display is the output device. Note that the data in the ACC are not altered due to the output, but an input operation replaces the original ACC content with the new data. Figure 4.1 shows the hardware components of ASC.

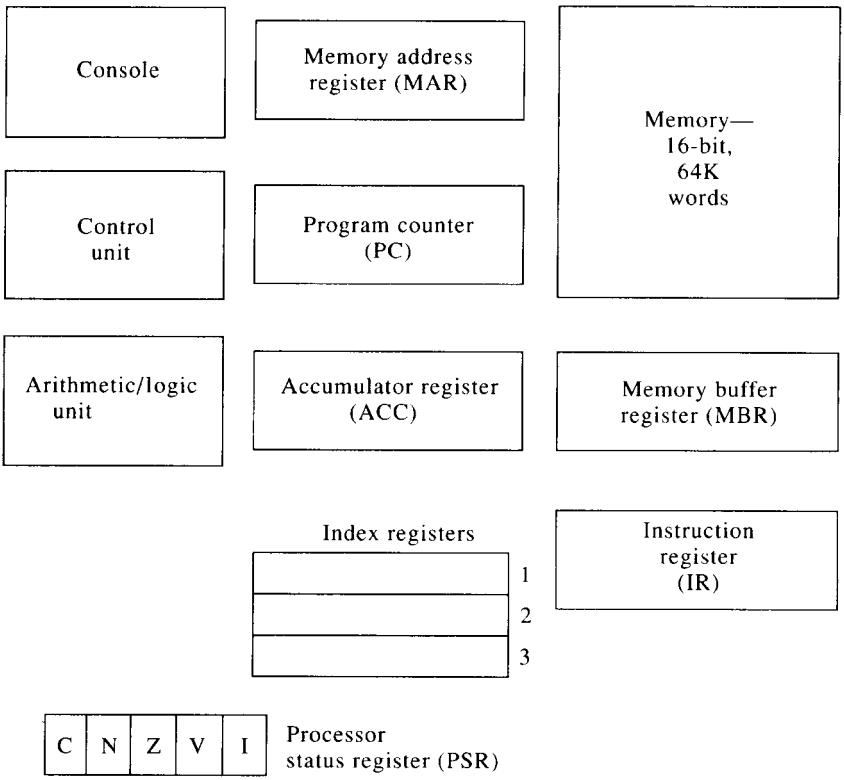



**Figure 4.1** ASC hardware components

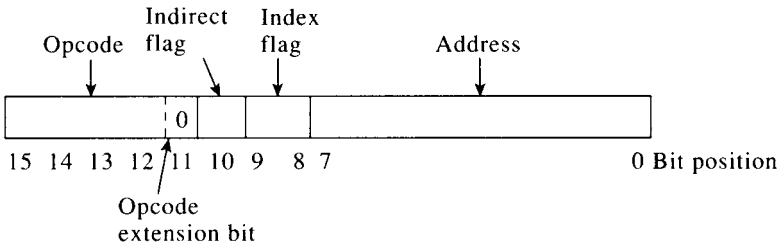
### 4.1.1 Data Format

ASC memory is an array of up to 64K 16-bit words. Each of these 16-bit words will be either an instruction or a 16-bit unit of data. The exact interpretation depends on the context in which the machine accesses a particular memory word. The programmer should be aware (at least at the assembly and machine-language programming levels) of the data and program segments in the memory and should make certain that a data word is not accessed during a phase in which the processor is accessing an instruction and vice versa.

Only fixed-point (integer) arithmetic is allowed on ASC. Figure 4.2 shows the data format: The most significant bit is the sign bit followed by 15 magnitude bits. Since ASC uses 2s complement representation, the sign and magnitude bits are treated alike in all computations. Note also the four-digit hexadecimal notation used to represent the 16-bit data word. We will use this notation to denote a 16-bit quantity irrespective of whether it is data or an instruction.

### 4.1.2 Instruction Format

Each instruction in an ASC program occupies a 16-bit word. An instruction word has four fields as shown below:




Bits 15 through 11 of the instruction word are used for the operation code (opcode). The *opcode* is a unique bit pattern that encodes a primitive operation the computer can perform. Thus, ASC can have a total of $2^5 = 32$ instructions. We will use an instruction set with only 16 instructions for simplicity in this book. The opcodes for these 16 instructions occupy bits 15 through 12, and bit 11 is set to 0. If the instruction set is to be expanded beyond the current set of 16, the opcodes for the new instructions would have a 1 in bit 11.

Bit 10 of the instruction word is the *indirect* flag. This bit will be set to 1 if indirect addressing is used; otherwise it is set to 0.

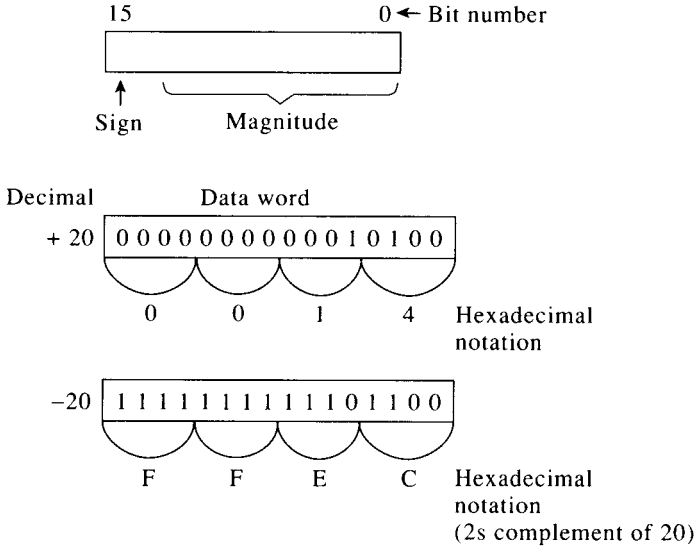

15
0 ← Bit number
Sign
Magnitude
Decimal
Data word
+ 20
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0 0 1 4
Hexadecimal notation
−20
1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0
F F E C
Hexadecimal notation (2s complement of 20)

**Figure 4.2** ASC data format

Bits 9 and 8 of the instruction word select one of the three index registers when indexed addressing is called for or if the instruction manipulates an index register:

| Bit 9 | Bit 8 | Index Register Selected |
|---|---|---|
| 0 | 0 | None |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

Bits 7 through 0 are used to represent the memory address in those instructions that refer to memory. If the instruction does not refer to memory, the indirect, index, and memory address fields are not used; the opcode field represents the complete instruction.

With only 8 bits in the address representation, ASC can directly address only $2^8 = 256$ memory locations. That means the program and data must always be in the first 256 locations of the memory. Indexed and indirect addressing modes are used to extend the addressing range to 64K. Thus ASC has direct, indirect, and indexed addressing modes. When both indirect and indexed addressing mode fields are used, the addressing mode can be interpreted either as indexed-indirect (preindexed indirect) or as indirect-indexed (postindexed indirect). We assume that ASC allows only

the indexed-indirect mode. We will describe the addressing modes further after the description of the instruction set that follows.

### 4.1.3 Instruction Set

Table 4.1 lists the complete instruction set of ASC. Column 2 shows the most significant four bits of the opcode in hexadecimal form. The fifth bit being 0 is not shown. Each opcode is also identified by a *symbolic name*, or *mnemonic*, shown in column 1.

We add one more construct to our HDL (hardware description language) described in Chapter 2. The memory is designed as $M$. A memory read operation is shown as

$$\text{MBR} \leftarrow \text{M}[MAR].$$

**Table 4.1** ASC Instruction Set

| Mnemonic | Opcode[1] (Hexadecimal) | Description |
|---|---|---|
| HLT | 0 | Halt |
| LDA | 1 | ACC $\leftarrow$ M[MEM][2] |
| STA | 2 | M[MEM] $\leftarrow$ ACC |
| ADD | 3 | ACC $\leftarrow$ ACC + M[MEM] |
| TCA | 4 | ACC $\leftarrow \overline{\text{ACC}}$ + 1 (2s complement) |
| BRU | 5 | Branch unconditional |
| BIP | 6 | Branch if ACC $>$ 0 |
| BIN | 7 | Branch if ACC $<$ 0 |
| RWD | 8 | Read a word into ACC |
| WWD | 9 | Write a word from ACC |
| SHL | A | Shift left ACC once |
| SHR | B | Shift right ACC once |
| LDX | C | INDEX $\leftarrow$ M[MEM] |
| STX | D | M[MEM] $\leftarrow$ INDEX |
| TIX | E | Test index increment |
| | | INDEX $\leftarrow$ INDEX + 1 |
| | | Branch if INDEX = 0 |
| | F | Test index decrement |
| TDX | | INDEX $\leftarrow$ INDEX – 1 |
| | | Branch if INDEX $\neq$ 0 |

1. Most significant four bits of opcode only. Bit 11 is 0.
2. MEM refers to a memory word; i.e., the symbolic address of a memory word. M[MEM] refers to the contents of the memory word MEM when used as a source and to the memory word when used as a destination.

and a memory write operation is shown as

M[*MAR*]← MBR.

The operand within the [ ] can be

1. A register; the content of the register is a *memory address*.
2. A symbolic address; the *symbolic address* will eventually be associated with an *absolute address*.
3. An absolute address.

Thus,

ACC ← M[27].

M[28] ← ACC.

IR ← M[Z1]. and

MZ1] ← ACC. (Z1 is a symbolic address.)

are all valid data transfers. Further,

ACC ← Z1.

implies that the absolute address value corresponding to Z1 is transferred to ACC. Thus,

Z1 ← ACC.

is not valid.

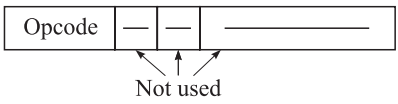
The ASC instruction set consists of the following three classes of instructions:

1. Zero address (TCA, HLT, SHL, and SHR).
2. One address (LDA, STA, ADD, BRU, BIP, BIN, LDX, STX, TIX, and TDX).
3. Input/output (RWD, WWD).

A description of instructions and their representation follows. In this description hexadecimal numbers are distinguished from decimal numbers with a preceding "#H."

## Zero-Address Instructions

In this class of instructions, the opcode represents the complete instruction. The operand (if needed) is implied to be in the ACC. The address field and the index and indirect flags are not used.



A description of each instruction follows:

| HLT Stop Halt |
|---|

The HLT instruction indicates the logical end of a program and hence stops the machine from fetching the next instruction (if any).

| TCA ACC ← ACC′ + 1 2s complement accumulator |
|---|

TCA complements each bit of the ACC to produce the 1s complement and then a 1 is added to produce the 2s complement. The 2s complement of the ACC is stored back into the ACC.

| SHL $ACC_{15-1} \leftarrow ACC_{14-0}$ Shift left<br>$ACC_0 \leftarrow 0$ |
|---|

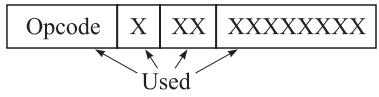
The SHL instruction shifts the contents of the ACC 1 bit to the left and fills a 0 into the least significant bit of the ACC.

| SHR $ACC_{14-0} \leftarrow ACC_{15-1}$ Shift right<br>$ACC_{15} \leftarrow ACC_{15}$ |
|---|

The SHR instruction shifts the contents of the ACC 1 bit to the right and the most significant bit of the ACC remains unchanged. The contents of the least significant bit position are lost.

### One-address Instructions

These instructions use all 16 bits of an instruction word. In the following, MEM is a *symbolic address* of an arbitrary memory location. An *absolute address* is the physical address of a memory location, expressed as a numeric quantity. A symbolic address is mapped to an absolute address when an assembly language problem is translated into machine language. The description assumes a direct addressing mode in which MEM is the *effective address* (the address of the operand). The 8-bit address is usually modified by the indirect and index operations to generate the effective address of a memory operand for each of these instructions.



The description of one-address instructions follows:

| LDA MEM ACC ← M[MEM]. Load accumulator |
|---|

LDA loads the ACC with the contents of the memory location (MEM) specified. Contents of MEM are not changed, but the contents of the ACC before the execution of this instruction are replaced by the contents of MEM.

| STA MEM M[MEM] ← ACC. Store accumulator |
|---|

STA stores the contents of the ACC at the specified memory location. ACC contents are not altered.

| ADD MEM ACC ← ACC + M[MEM]. Add |
|---|

ADD adds the contents of the memory location specified to the contents of the ACC. Memory contents are not altered.

| BRU MEM PC ← MEM. Branch unconditional |
|---|

BRU transfers the program control to the address MEM. That is, the next instruction to be executed is at MEM.

| BIP MEM IF ACC > 0 THEN PC ← MEM Branch if ACC is positive |
|---|

The BIP instruction tests the N and Z bits of PSR. If both of them are 0, then the program execution resumes at the address (MEM) specified; if not, execution continues with the next instruction in sequence. Since the PC must contain the address of the instruction to be executed next, the branching operation corresponds to transferring the address into PC.

| BIP MEM IF ACC < 0 THEN PC ← MEM. Branch if accumulator negative |
|---|

The BIN instruction tests the N bit of PSR; if it is 1, program execution resumes at the address specified; if not, the execution continues with the next instruction in sequence.

| LDX MEM, INDEX INDEX ← M[MEM]. Load index register |
|---|

The LDX loads the index register (specified by INDEX) with the contents of memory location specified. In the assembly language instruction format, INDEX will be 1, 2, or 3.

| | | | |
|---|---|---|---|
| STX | MEM, INDEX | M[MEM] ← INDEX. | Store index register |

The STX stores a copy of the contents of the index register specified by the index flag into the memory location specified by the address. The index register contents remain unchanged.

| | | | |
|---|---|---|---|
| TIX | MEM, INDEX | INDEX ← INDEX + 1 | Test index |
| | | IF INDEX = 0 THEN PC ← MEM. | increment |

TIX increments the index register content by 1. Next, it tests the index register content; if it is 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

| | | | |
|---|---|---|---|
| TDX | MEM, INDEX | INDEX ← INDEX − 1 | Test index |
| | | IF INDEX ≠ 0 THEN PC ← MEM. | decrement |

TDX decrements the index register content by 1. Next it tests the index register content; if it is not equal to 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

It is important to note that LDX, STX, TDX, and TIX instructions "refer" to an index register as one of the operands. Indexed mode of addressing is thus not possible with these instructions since the index field is used for the index register reference. Only direct and indirect modes of addressing can be used. For example:

LDA Z, 3 adds the contents of index register 3 to Z to compute the effective address EA. Then the contents of memory location EA are loaded into the ACC. Index register is not altered; whereas,

LDX Z, 3 loads the index register 3 from the contents of memory location Z.

### Input/Output Instructions

Since ASC has one input and one output device, the address, index, and indirect fields in the instruction word are not used. Thus, these are also zero-address instructions.

| RWD ACC ← Input data. Read a word |
|---|

RWD instruction reads a 16-bit word from the input device into the ACC. The contents of the ACC before RWD are thus lost.

| WWD Output ← ACC. Write a word |
|---|

WWD instruction writes a 16-bit word from the ACC onto the output device. ACC contents remain unaltered.

## 4.1.4 Addressing Modes

Addressing modes allowed by a machine are influenced by the programming languages and corresponding data structures that the machine uses. ASC instruction format allows the most common addressing modes. Various other modes are used in machines commercially available. They are described in Chapter 7.

ASC addressing modes are described below with reference to the load accumulator (LDA) instruction. Here, Z is assumed to be the symbolic address of the memory location 10. For each mode, the assembly language format is shown first, followed by the instruction format encoded in binary (i.e., the machine language). The *effective address* calculation and the effect of the instruction are also illustrated. Note that the effective address is the address of the memory word where the operand is located.

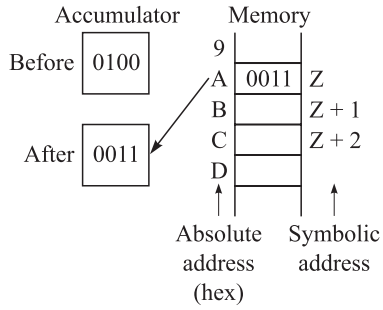
### Direct Addressing

Instruction format: LDA Z

| 00010 | 0 | 00 | 00001010 |
|---|---|---|---|

Effective address: Z or #HA
Effect: ACC ← M[Z].

The effect of this instruction is illustrated below. We will use hexadecimal notation to represent all data and addresses in the following diagrams.




Note: Contents of register and memory are shown in hexadecimal.

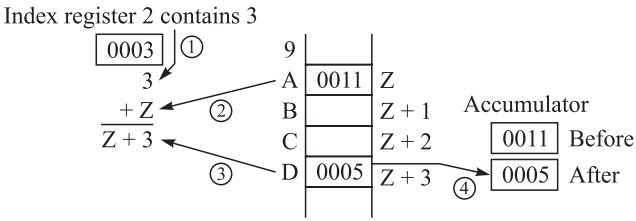
## Indexed Addressing

Instruction format: LDA Z,2

| 00010 | 0 | 10 | 00001010 |
|---|---|---|---|

Effective address: Z + index register 2
Effect: ACC ← M[Z + index register 2].

The number in the operand field after the comma denotes the index register used. Assuming that index register 2 contains 3, the following diagram illustrates the effect of this instruction. The numbers in circles show the sequence of operations. Contents of index register 2 are added to Z to derive the effective address Z + 3. Contents of location Z + 3 are then loaded into the accumulator.




Note that the *address field* of the instruction refers to Z and the contents of the index register specify an offset from Z. Contents of an index register can be varied by using LDX, TIX, and TDX instructions, thereby accessing various memory locations as offsets of Z. Indexing in this way allows us to access consecutive memory locations dynamically, by changing the contents of the index register. Further, since index registers are 16 bits wide, the effective address can be 16 bits long, thereby extending the mem-

ory addressing range to 64K from the range of 256 locations possible with 8 address bits.

The most common use of indexed addressing mode is in referencing the elements of an array. The address field in the instruction points to the first element. Subsequent elements are referenced by incrementing the index register.

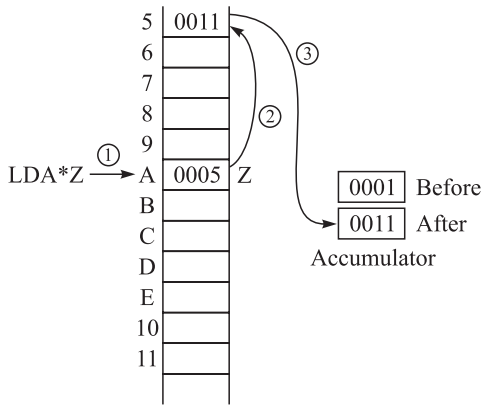
### Indirect Addressing

Instruction format: LDA* Z

| 00010 | 1 | 00 | 0000 1010 |
|---|---|---|---|

Effective address: M[Z]
Effect: MAR ← M[Z].
ACC ← M[MAR].
i.e., ACC ← M[M[Z]].

The asterisk next to the mnemonic denotes the indirect addressing mode. In this mode, the address field points to a location where the address of the operand can be found.




Since a memory word is 16 bits long, the indirect addressing mode can also be used to extend the addressing range to 64K. Further, by simply changing the contents of location Z in the above illustration, we can refer to various memory addresses using the same instruction. This feature is useful, for example, in creating a multiple jump instruction in which contents of Z are dynamically changed to refer to the appropriate address to jump to. The most common use of indirect addressing is in referencing data

elements through pointers. A pointer contains the address of the data to be accessed. The data access takes place through indirect addressing, using the pointer as the operand. When data are moved to other locations, it is sufficient to change the pointer value accordingly, in order to access the data from the new location.

If both indirect and index flags are used, there are two possible modes of effective address computation, depending on whether indirecting or indexing is performed first. They are illustrated below.

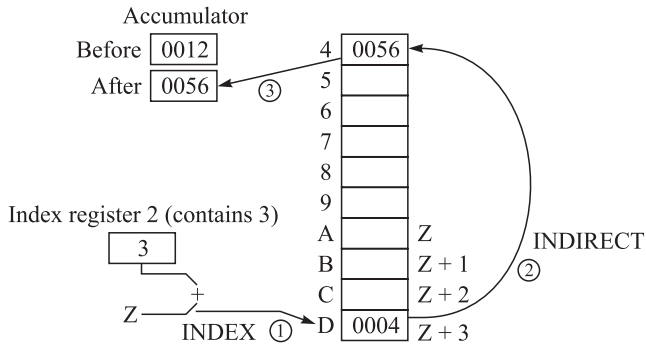
### Indexed-indirect Addressing (preindexed-indirect)

Instruction format: LDA* Z,2

| 00010 | 1 | 10 | 0000 1010 |
|---|---|---|---|

Effective address: M[Z + index register 2]
Effect: ACC ← M[M[Z + index register 2]].

Indexing ① is done first, followed by indexing ② to compute the effective address whose contents are loaded ③ into the accumulator as shown:



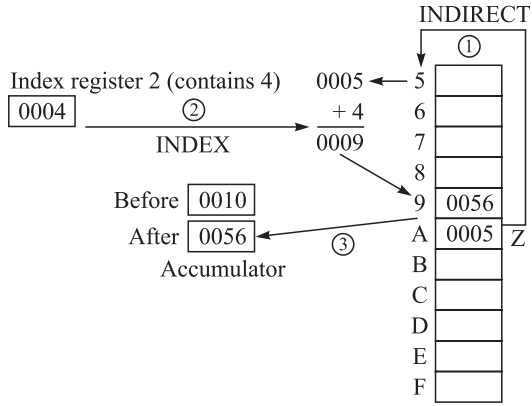

### Indirect-indexed Addressing (postindexed-indirect)

Instruction format: LDA* Z,2

| 00010 | 1 | 10 | 00001010 |
|---|---|---|---|

Effective address: M[Z] + index register 2
Effect: ACC ← M[M[Z] + index register 2].

Indirection ① is performed first, followed by indexing ② to compute the effective address whose contents are loaded ③ into the accumulator.




Note that the instruction formats in the above two modes are identical. ASC cannot distinguish between these two modes. The indirect flag must be expanded to 2 bits if both of these modes are to be allowed. Instead, we assume that *ASC always performs preindexed-indirect and post-indexing is not supported.*

The above addresing modes are applicable to all single-address instructions. The only exceptions are the index-reference instructions (LDX, STX, TIX, and TDX) in which indexing is not permitted.

Consider an array of pointers located in consecutive locations in the memory. The preindexed-indirect addresing mode is useful in accessing the data elements since we can first index to a particular pointer in the array and indirect on that pointer to access the data element. On the other hand, the postindexed-indirect mode is useful in setting up pointers to an array since we can access the first element of the array by indirecting on the pointer and access subsequent elements of the array by indexing over the pointer value.

### 4.1.5 Other Addressing Modes

Chapter 7 describes several other addressing modes that are employed in practice. For example, it is convenient sometimes in programming to include data as part of the instruction. *Immediate addressing mode* is used in such cases. Immediate addressing implies that the data are part of the instruction itself. *This mode is not allowed in ASC*, but the instruction set can be extended to include instructions such as load immediate (LDI), add immediate (ADI), etc. In such instructions, the opcode field will contain a 5-bit opcode, and the remaining 11 bits will contain the data. For instance,

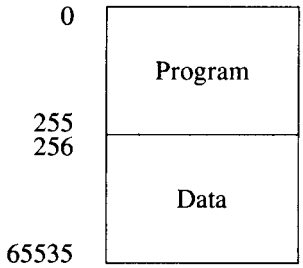
LDI 10 would imply loading 10 into ACC, and
ADI 20 would imply adding 20 to the ACC.

Since ASC does not permit this addressing mode, the ASC assembler is designed to accept the so-called *literal addressing mode*, which simulates the immediate addressing mode on ASC. Refer to the next section in this chapter for further details.

### 4.1.6 Addressing Limitations

As discussed earlier, ASC instruction format restricts the direct-addressing range to the first 256 locations in the memory. Thus, if the program and data can fit into locations 0 through 255, no programming difficulties are encountered. If this is not possible, the following programming alternatives are possible:

1. The program resides in the first 256 locations, and the data resides in higher-addressed locations in the memory.




   In this case, all instruction addresses can be represented by the 8-bit address field. Since data references require an address field longer than 8 bits, all data references are handled using indexed and indirect addressing modes. For example, the data location 300 can be loaded into the ACC by either of the following instructions:

   LDA 0,2 assuming that index register 2 contains 300.
   LDA* 0 assuming that location 0 in the memory contains 300.

2. Data resides in the first 256 locations, and the program resides beyond location 255. In this case, all data reference instructions (such as LDA, STA, etc.) can use direct, indirect, and/or indexed modes, but all other memory reference instructions (such as BRU, BIP, BIN) must use indexed and/or indirect modes.
3. If the program and data both reside beyond location 255, all memory reference instructions must be indirect and/or indexed.

Recall that the index reference instructions can use only the indirect mode of addressing.

### 4.1.7 Machine Language Programming

It is possible to write a program for ASC using absolute addresses (actual physical memory addresses) and opcodes only, since the instruction set and instruction and data formats are now known. Such programs are called *machine language programs* They need not be further translated for the hardware to interpret them since they are already in binary form. Programming at this level is tedious, however. A program in machine language to add two numbers and store the sum in a third location is the following:

```
0001 0000 0000 1000
0011 0000 0000 1001
0010 0000 0001 0000
```

Can you decode these instructions and determine what the program is doing?

Modern-day computers are seldom programmed at this level. All programs must be at this level, however, before execution of the program can begin. Translators (assemblers and compilers) are used in converting programs written in assembly and high-level languages into this machine language. We will discuss a hypothetical assembler for ASC assembly language in the next section.

## 4.2 ASC ASSEMBLER

An assembler that translates ASC assembly language programs into machine language programs is available. We will provide details of the language as accepted by this assembler and outline the assembly process in this section.

An assembly language program consists of a sequence of statements (instructions) coded in mnemonics and symbolic addresses. Each statement consists of four fields: label, operation (mnemonic), operand, and comments, as shown in Fig. 4.3. The *label* is a symbolic name denoting the memory location where the instruction itself is located. It is not necessary to provide a label for each statement. Only those statements that are referenced from elsewhere in the program need labels. When provided, the label is a set of alphabetic and numeric characters, the first of which must be an alphabetic character. The *mnemonic* field contains the instruction mnemo-

| Label[1] | Mnemonic | Operand | Comments[1] |
|---|---|---|---|
| Consists of alphabetic and numeric characters.<br>First character must be alphabetic.<br>An "*" as first character denotes that complete statement is a comment. | Comprises three-character standard symbolic opcodes.<br>An "*" as fourth character signifies indirect addressing. | Consists of absolute and symbolic addresses.<br>Indexes register designations following a ",". | Starts with a ".". Assembler ignores it. |

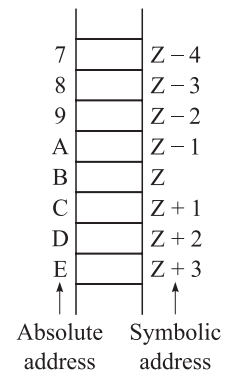
[1]Optional field.
Note: A space (partition) is required between label and mnemonic fields and between mnemonic and operand fields.

**Figure 4.3** Assembly language statement format

nic. An "*" following the instruction mnemonic denotes indirect addressing. The *operand* field consists of symbolic addresses, absolute addresses, and index register designations. Typical operands are:

| OPERAND | DESCRIPTION |
|---|---|
| 25 | Decimal 25 (by default) |
| #H25 | Hexadecimal 25 |
| #O25 | Octal 25 |
| #B1001 | Binary 1001 |
| Z | Symbolic address Z |
| Z, 1 | Z indexed with index register 1 |
| Z + 4 | Four locations after Z |
| Z + 4, 1 | Address Z + 4 indexed with register 1 |
| Z – 4 | Address Z – 4 (four locations before Z) |
| Z – P | Z and P are symbolic addresses;<br>Z – P yields an absolute address;<br>Z must be at a higher physical address than P for Z – P value to be positive. |

MEMORY




The *comments* fields starts with a ".". This optional field consists only of comments by the programmer. It does not affect the instruction in any way and is ignored by the assembler. An "*" as the first character in the label field designates that the complete statement is a comment.

Each instruction in an assembly language program can be classified as either an *executable instruction* or an *assembler directive* (or a pseudo-instruction). Each of the sixteen instructions in ASC instruction set is an executable instruction. The assembler generates a machine language instruc-

tion corresponding to each such instruction in the program. A *pseudo-instruction* is a directive to the assembler. This instruction is used to control the assembly process, to reserve memory locations, and to establish constants required by the program. The pseudo-instructions when assembled do not generate machine language instructions and as such are not executable. Care must be taken by the assembly language programmer to partition the program such that an assembler directive is not in the execution sequence. A description of ASC pseudo-instructions follows:

| ORG Address Origin |
|---|

The function of ORG directive is to provide the assembler the memory address where the next instruction is to be located. ORG is usually the first instruction in the program. Then the operand field of ORG provides the starting address (i.e., the address where the first instruction in the program is located). If the first instruction in the program is not ORG, the assembler defaults to a starting address of 0. There can be more than one ORG directive in a program.

| END Address Physical end |
|---|

END indicates the physical end of the program and is the last statement in a program. The operand field of the END normally contains the label of the first executable statement in the program.

| EQU Equate |
|---|

EQU provides a means of giving multiple symbolic names to memory locations, as shown by the following example:

---

**Example 4.1**

```
A   EQU  B        A is another name for B. (B must already be defined.)
A   EQU  B + 5    A is the name of location B + 5.
A   EQU  10       A is the name of the absolute address 10.
```

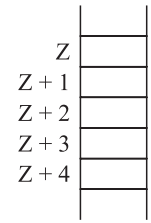
---

| BSS Block storage starting |
|---|

BSS is used to reserve blocks of storage locations for intermediate or final results

**Example 4.2**

```
Z   BSS   5   Reserves fives loactions,
              the first of which is
              named Z.
```




The operand field always designates the number of locations to be reserved. Contents of these reserved locations are not defined.

BSC Block storage of constants

BSC provides a means of storing constants in memory locations in addition to reserving those locations. The operand field consists of one or more operands (separated by a ","). Each operand requires one memory word.

**Example 4.3**

```
Z   BSC 5          Reseves one location named Z containing a 5.
P   BSC 5, –6, 7   Reserves three locations: P containing 5, P + 1 contain-
                   ing –6, and P + 2 containing 7.
```

## Literal Addressing Mode

It is convenient for the programmer to be able to define constants (data) as a part of the instruction. This feature also makes an assembly language program more readable. Literal addressing mode enables this. A literal is a constant preceded by an "=". For example,

```
LDA = 2
```

implies loading a constant 2 (decimal) into the accumulator, and

```
ADD = #H10
```

implies adding a #H10 to the accumulator. ASC assembler recognizes such literals, reserves an available memory location for the constant in the address field, and substitutes the address of the memory location into the instruction.

We will now provide some assembly language programs as examples.

---

**Example 4.4** Figure 4.4 shows a program to add three numbers located at A, B, and C and save the result at D. The program is ORGined to location 10. Note how the HLT statement separates the program logic from the data block. A, B, and C are defined using BSC statements, and one location is reserved for D using a BSS.

---

---

**Example 4.5** Figure 4.5 shows a program to accumulate 5 numbers stored starting at location X in memory and store the result at Z. Here, index register 1 is first set to 4 so as to point to the last number in the block (i.e., at X + 4). The ACC is set to 0. TDX is used to access numbers one at a time from the last to first and to terminate the loop after all the numbers are accumulated.

---

```
*   PROGRAM TO ADD 3 NUMBERS
            ORG   10
BEGIN       LDA   A
            ADD   B
            ADD   C
            STA   D
            HLT
A           BSC   5
B           BSC   7
C           BSC   -3
D           BSS   1
            END BEGIN
```

Note: BEGIN points to the first executable instruction in the program, and the operand of the END instruction is BEGIN.

**Figure 4.4** A program to add 3 numbers

```
*  A PROGRAM TO ACCUMULATE THE VALUES OF 5
*   NUMBERS LOCATED AT X

           ORG  0
START      LDX  =4,1
           LDA  =0                .Zero accumulator
LOOP       ADD  X,1               . ADD loop
           TDX  LOOP,1
           ADD  X                 .ADD the remaining value
           STA  Z                 . Store the result in Z
           HLT
X          BSC  5,35,26,-7,4
Z          BSS  1
           END  START
```

**Figure 4.5** A program to accumulate a block of numbers

---

**Example 4.6** *Division*. Division can be treated as the repeated subtraction of the divisor from the dividend until a zero or negative result is obtained. The quotient is equal to the maximum number of times the subtraction can be performed without yielding a negative result. Figure 4.6 shows the division routine.

---

The generation of the object code from the assembly language programs is described next.

### 4.2.1 Assembly Process

The major functions of the assembler program are (1) to generate an address for each symbolic name in the program and (2) to generate the binary equivalent of each assembly instruction. The assembly process is usually carried out in two scans over the source program. Each of these scans is called a *pass* and the assembler is called a *two-pass assembler*. The first pass is used for allocating a memory location for each symbolic name used in the program; during the second pass, references to these symbolic names are resolved. If the restriction is made that each symbol must be defined before it can be referenced, one pass will suffice.

Details of the ASC two-pass assembler are given in Figs 4.7 and 4.8. The assembler uses a counter known as *location counter* (LC) to keep track of the memory locations used. If the first instruction in the program is ORG, the operand field of ORG defines the initial value of LC; otherwise, LC is set

```
0001                *       SOURCE PROGRAM
0002                *                          . A DIVISION ALGORITHM FOR ASC
                                                 (A/B)
0003                *                          . A AND B ARE NON-
                                                 NEGATIVE INTEGERS
0004                        ORG   0
0005  0000  8000  BEGIN    RWD                 . READ A
0006  0001  201E           STA   A
0007  0002  6004           BIP   NEXT
0008  0003  501B           BRU   OUT           . IF A=0, QUOTIENT=0
0009  0004  8000  NEXT     RWD                 . READ B
000A  0005  201F           STA   B
000B  0006  6008           BIP   INIT
000C  0007  501B           BRU   OUT           . IF B=0, THEN BY DEFN QUO=0
000D  0008  1021  INIT     LDA   ZERO          . INITIALIZE QUOTIENT TO 0
000E  0009  2020           STA   COUNT
000F  000A  101F           LDA   B
0010  000B  4000           TCA
0011  000C  201F           STA   B             . STORE NEGATIVE B IN B
0012  000D  101F  LOOP     LDA   B
0013  000E  301E           ADD   A             . A-(I*B), I=NUMBER OF LOOPS
0014  000F  201E           STA   A
0015  0010  7015           BIN   FINISH        . RESULT NEGATIVE, FINISHED
0016                *                          . DO NOT INCREMENT COUNTER
0017  0011  6017           BIP   OTHER
0018  0012  1020           LDA   COUNT         . REMAINDER IS ZERO
0019  0013  3022           ADD   ONE
001A  0014  501C           BRU   OUT1
001B  0015  1020  FINISH   LDA   COUNT         . LOAD FINAL COUNT
001C  0016  501C           BRU   OUT1
001D  0017  1020  OTHER    LDA   COUNT         . INCREMENT QUOTIENT
001E  0018  3022           ADD   ONE           . RESULT IS 0 OR POSITIVE
001F  0019  2020           STA   COUNT
0020  001A  500D           BRU   LOOP
0021  001B  1021  OUT      LDA   ZERO
0022  001C  9000  OUT1     WWD                 . WRITE RESULT
0023  001D  0000           HLT
0024  001E  0000  A        BSS   1
0025  001F  0000  B        BSS   1
0026  0020  0000  COUNT    BSS   1
0027  0021  0000  ZERO     BSC   0
0028  0022  0001  ONE      BSC   1             . END
0029                        END   BEGIN
```

Note: The assembler is assumed to assemble 0s for undefined bits in BSS and zero-address instructions.

**Figure 4.6** ASC division program

to 0. LC is incremented appropriately during the assembly process. Content of LC at any time is the address of the next available memory location.

The assembler performs the following tasks during the first pass:

1. Enters labels into a symbol table along with the LC value as the address of the label.
2. Validates mnemonics.

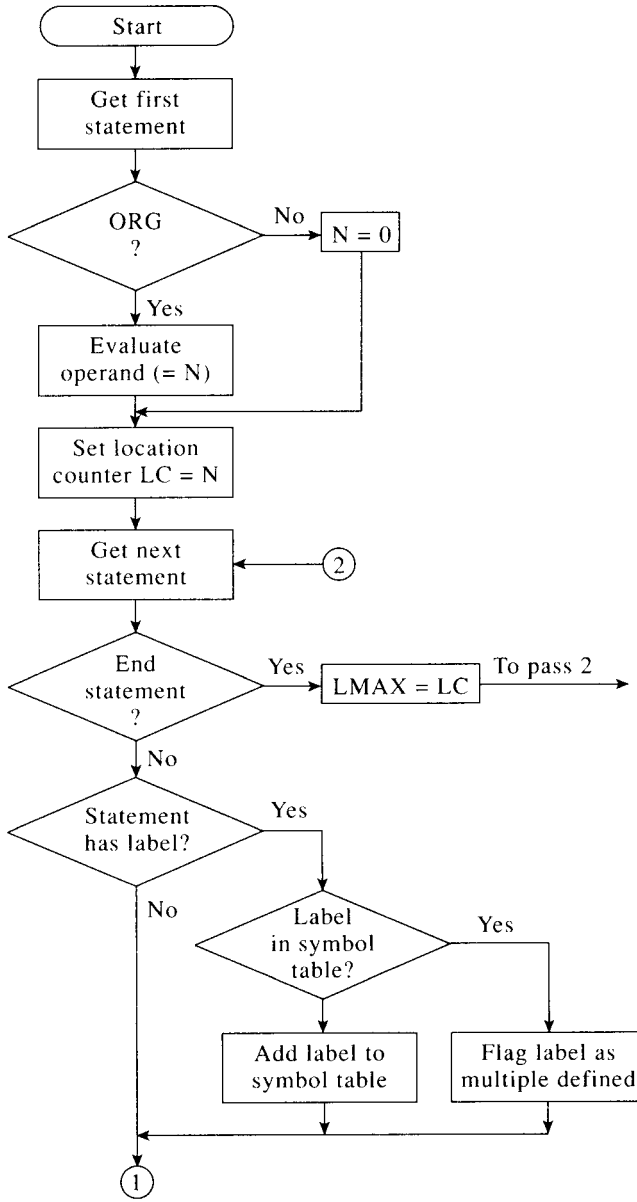



**Figure 4.7** Assembler pass 1

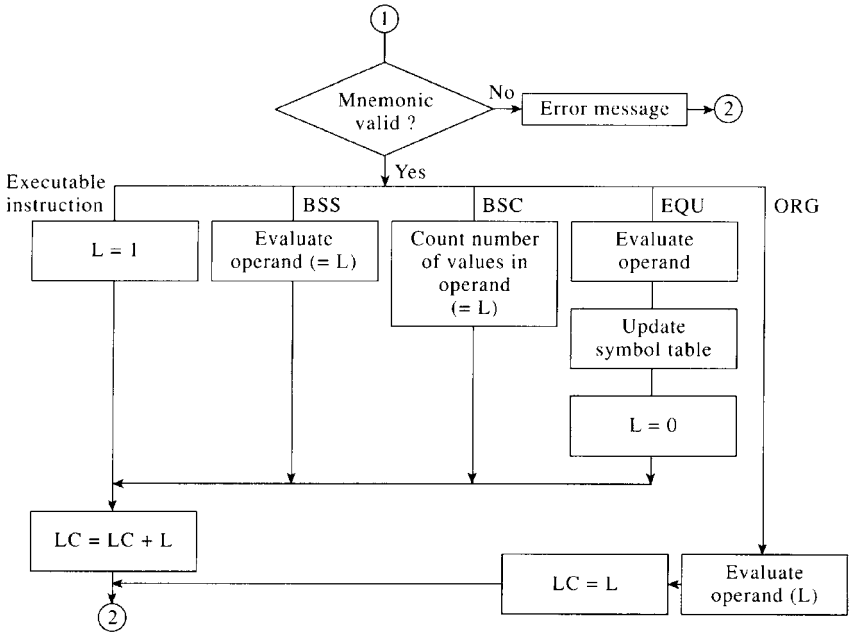



**Figure 4.7** (Continued)

3. Interprets pseudo-instructions completely.
4. Manages the location counter.

The major activities during the second pass are:

1. Evaluation of the operand field.
2. Insertion of opcode, address, and address modifiers into the instruction format.
3. Resolution of literal addressing.

The assembler uses an opcode table to extract opcode information. The opcode table is a table storing each mnemonic, the corresponding opcode, and any other attribute of the instruction useful for the assembly process. The symbol table created by the assembler consists of two entries for each symbolic name: the symbol itself and the address in which the symbol will be located. We will illustrate the assembly process in Example 4.7.

---

**Example 4.7** Consider the program shown in Fig. 4.9(a). The symbol table is initially empty. Location counter starts at the default value of 0. The first instruction is ORG. Its operand field is evaluated and the value (0) is entered

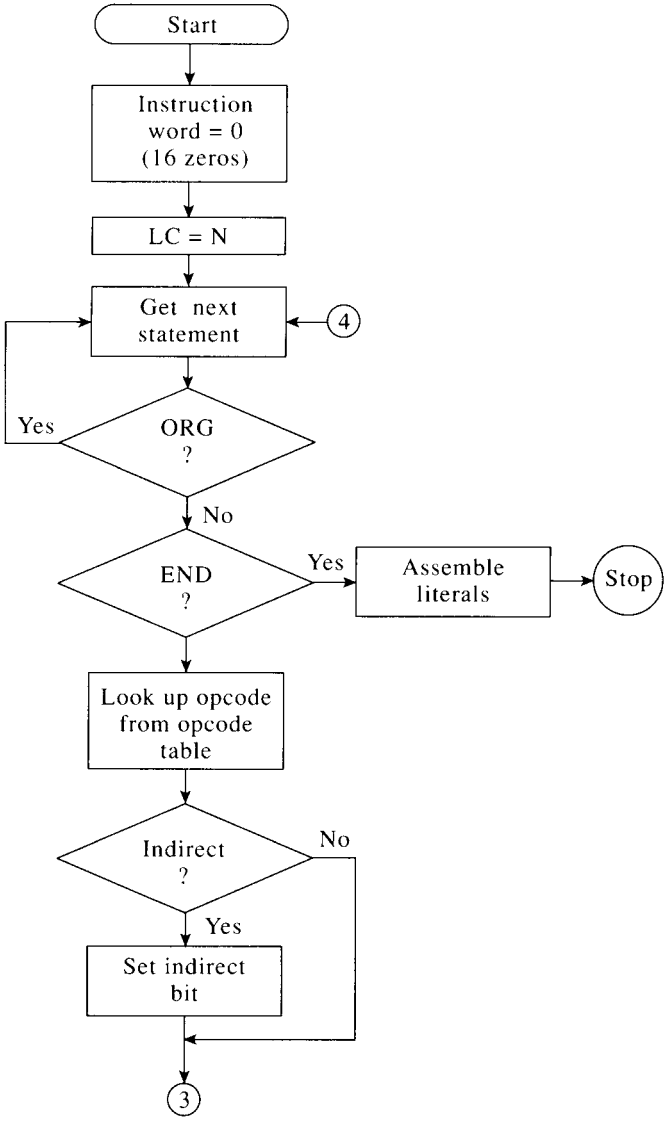



**Figure 4.8** Assembler pass 2

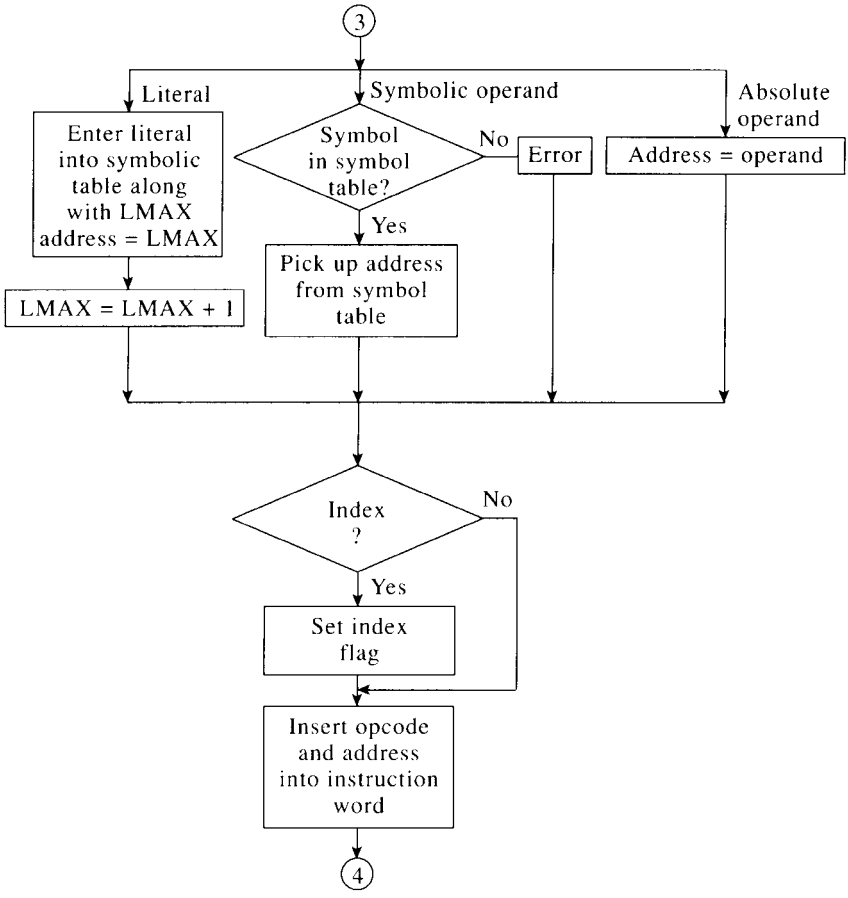
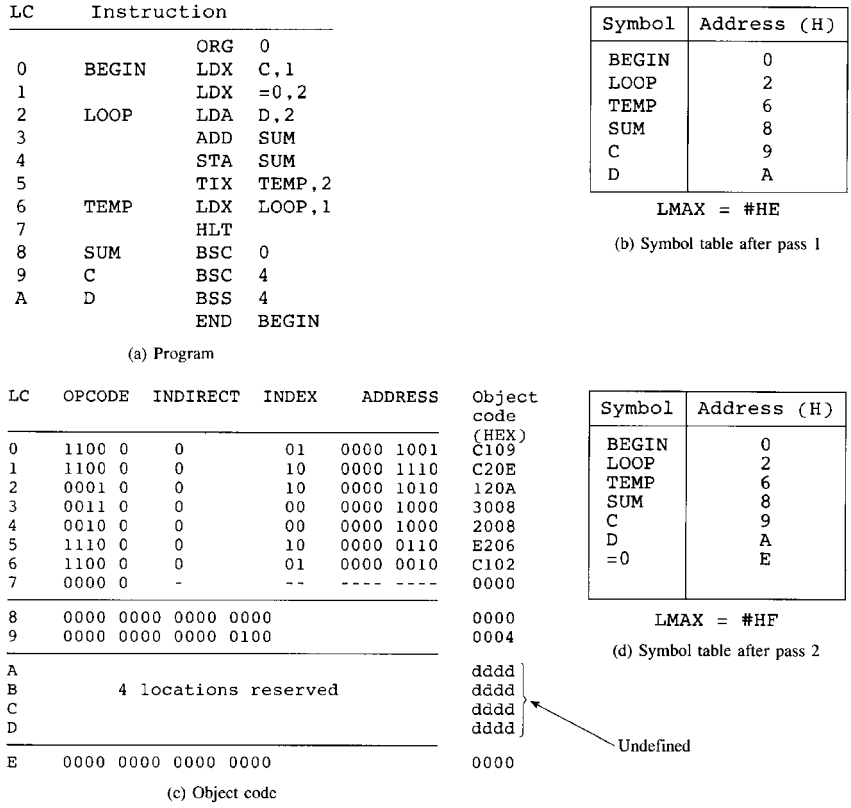



**Figure 4.8** (Continued)

into LC. The label field of the next instruction is BEGIN. BEGIN is entered into symbol table and assigned the address of 0. The mnemonic field has LDX, which is a valid mnemonic. Since this instruction takes up one memory word, LC is incremented by 1. The operand field of LDX instruction is not evaluated during the first pass. This process of scanning the label and mnemonic fields, entering labels (if any) into the symbol table, validating mnemonic, and incrementing LC continues until END instruction is reached. Pseudo-instructions are completely evaluated during this pass. Location counter values are shown in Fig. 4.9(a) along with symbol table entries at the end of pass 1 in (b). By the end of the first pass, the location



```
LC     Instruction
                ORG   0
0      BEGIN    LDX   C,1
1               LDX   =0,2
2      LOOP     LDA   D,2
3               ADD   SUM
4               STA   SUM
5               TIX   TEMP,2
6      TEMP     LDX   LOOP,1
7               HLT
8      SUM      BSC   0
9      C        BSC   4
A      D        BSS   4
                END   BEGIN
```

(a) Program

| Symbol | Address (H) |
|---|---|
| BEGIN | 0 |
| LOOP | 2 |
| TEMP | 6 |
| SUM | 8 |
| C | 9 |
| D | A |

LMAX = #HE

(b) Symbol table after pass 1

```
LC   OPCODE   INDIRECT  INDEX    ADDRESS     Object code (HEX)
0    1100     0        0        01   0000 1001   C109
1    1100     0        0        10   0000 1110   C20E
2    0001     0        0        10   0000 1010   120A
3    0011     0        0        00   0000 1000   3008
4    0010     0        0        00   0000 1000   2008
5    1110     0        0        10   0000 0110   E206
6    1100     0        0        01   0000 0010   C102
7    0000     0        -        --   ----  ----   0000
8    0000 0000 0000 0000                          0000
9    0000 0000 0000 0100                          0004
A                                                 dddd
B          4 locations reserved                   dddd
C                                                 dddd
D                                                 dddd
E    0000 0000 0000 0000                          0000
```

(c) Object code

| Symbol | Address (H) |
|---|---|
| BEGIN | 0 |
| LOOP | 2 |
| TEMP | 6 |
| SUM | 8 |
| C | 9 |
| D | A |
| =0 | E |

LMAX = #HF

(d) Symbol table after pass 2



**Figure 4.9** Assembly process

counter will have advanced to E, since BSS 4 takes up four locations (A through D).

During the second pass, machine instructions are generated using the source program and the symbol table. The operand fields are evaluated during this pass and instruction format fields are appropriately filled for each instruction. Starting with LC = 0, the label field of instruction at 0 is ignored and the opcode (11000) is substituted for the mnemonic LDX. Since this is a one-address instruction, the operand field is evaluated. There is no "*" next to the mnemonic, and hence the indirect flag is set to 0. The absolute address of C is obtained from the symbol table and entered into the address field of the instruction, and the index flag is set to 01. This process continues for each instruction until END is reached. The object code is shown in Fig. 4.9(c) in binary and hexadecimal formats. The symbol table shown in (d) has one

more entry corresponding to the literal = 0. Note that the instruction LDX = 0,2 has been assembled as LDX#HE, 2, with the location #HE containing a 0. Contents of the words reserved in response to BSS are not defined, and unused bits of HLT instruction words are assumed to be 0s.

## 4.3 PROGRAM LOADING

The object code must be loaded into the machine memory before it can be executed. ASC console can be used to load programs and data into the memory. Loading through the console is tedious and time consuming, however, especially when programs are large. In such cases, a small program that reads the object code statements from the input device and stores them in appropriate memory locations is first written, assembled, and loaded (using the console) into machine memory. This loader program is then used to load the object code or data into the memory.

Figure 4.10 shows a loader program for ASC. Instead of loading this program each time through the console, it can be stored in a ROM that forms part of the 64K memory space of ASC. Then loading can be initiated by setting the PC to the beginning address of the loader (using the console) and starting the machine. Note that the loader occupies locations 244 through 255. Hence, care must be taken to make sure that other programs do not overwrite this space.

```
Label    Mnemonic   Operand    Comment
         ORG        244
START    RWD                   .Read starting address
         STA        SAVE
         LDX        SAVE,2     .Starting address in index register 2
         RWD                   .Input number of statements N
         STA        SAVE       .SAVE contains N
         LDX        SAVE,1     .Load (N) into index register 1
LOOP     RWD                   .Input an object code statement
         STA        0,2        .Store the object code statement
LL       TIX        LL,2       .Increment index register 2
         TDX        LOOP,1     .Decrement N by 1 and loop
         HLT
SAVE     BSS        1
         END        START
```

**Figure 4.10** A loader for ASC

Note also that the assembler itself must be in the binary object code form and loaded into ASC memory before it can be used to assemble other programs. That means the assembler must be translated from the source language to ASC binary code either manually or by implementing the assembler on some other machine. It can then be loaded into ASC either by using the loader or by retaining it in a ROM portion of the memory for further use.

## 4.4 SUMMARY

This chapter is a programmer's introduction to ASC organization. Various components of ASC have been assumed to exist, and no justification has been given as to why a component is needed. Architectural tradeoffs used in selecting the features of a machine are discussed in subsequent chapters of the book. Chapter 5 provides the detailed hardware design of ASC. Details of an assembler for ASC, assembly language programming, and the assembly process have been described. A brief introduction to program loaders has been given. Further details on these topics can be found in the following references.

## REFERENCES

Baron, R. J. and Higbie, L. *Computer Architecture*, Reading, MA: Addison Wesley, 1992.

Calingaert, P. *Assemblers, Compilers and Program Translation*, Rockville, MD: Computer Science Press, 1979.

Mano, M. M. and Kime, C. R. *Logic and Computer Design Fundamentals*, Englewood Cliffs, NJ: Prentice-Hall, 1996.

Silberschatz, A. and Galvin, P. B. *Operating System Concepts*, Reading, MA: Addison Wesley, 1997.

Tannenbaum, A. S. and Goodman, J. R. *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall, 1998.

Tannenbaum, A. S., Woodhull, A. S. and Woodhull, A. *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1997.

Watt, D. A. *Programming Language Processors: Compilers and Interpreters*, Englewood Cliffs, NJ: Prentice-Hall, 1993.

## PROBLEMS

4.1 Calculate the effective address for each of the following instructions. (Remember: An "*" as the fourth symbol denotes indirect addressing.)

```
STA     Z
STA     Z,3
STA*    Z
STA*    Z,3     (Pre-index)
STA*    Z,3     (Post-index)
```

Assume index register 3 contains 5: Z is memory location #H10 and is the first location of a memory block containing 25, 7, 4, 86, 46, and 77.

4.2 Determine the contents of the ACC after each of the following instructions, assuming that the ACC is set to 25 before each instruction is executed:

```
SHL
ADD  =5
TCA
ADD  =#H3
SHR
```

4.3 Determine the content of the ACC at the end of each instruction in the above problem, assuming that the ACC is set to 25 to begin with and the effect of each instruction is cumulative; i.e., the ACC is not reset to 25 at the beginning of second and subsequent instructions.

4.4 What is the difference (if any) between:
a. A location counter and a program counter
b. END and HLT
c. BSS and BSC
d. ORG and EQU
e. Executable and pseudo instructions.

4.5 Show the contents (in Hexadecimal) of the memory locations reserved by the following program:

```
    ORG 200
    BSS  3
Z   BSC  3, −3, 5
    BSS  24
M   EQU Z + 1
    END
```

4.6 What is the effect of inserting ORG 300 instruction after the instruction at Z, in the program of Problem 4.5?

4.7 There are two ways of loading the accumulator with a number:

```
    LDA    X          LDI  =5
X   BSC    5
```

a. Which of the two executes faster? Why?
b. What is the range of the data value that can be loaded by each method?

4.8 NOP (No operation) is a common instruction found in almost all instruction sets. How does the assembler handle this instruction? Why is it needed?

4.9 Another popular addressing mode is PC-relative addressing. An example of such an instruction is BRU +5, meaning add 5 to the current PC value to determine the target address. Similarly, the target address of BRU –5 is determined by subtracting 5 from the current PC value.

a. How does the assembler handle this instruction?
b. What is the range of relative jump addresses possible in ASC?
c. What are the advantages and disadvantages of this mode compared to other jump addressing modes?

4.10 Give the symbol table resulting from the assembly of the following program.

```
        ORG    200
BEGIN   RWD
        STA    A
        BIP    TEMP 1
        BIN    TEMP 1
        BRU    OUT
TEMP1   LDX    B,1
        LDX    =0, 2
LOOP    LDA    C, 2
        ADD    SUM
        STA    SUM
        TIX    TEMP2, 2
TEMP2   TDX    OUT, 1
        BRU    LOOP
OUT     HLT
        ORG    500
SUM     BSC    0
A       BSS    1
B       BSC    4
C       BSS    4
        END    BEGIN
```

4.11 The dump of locations 64 to 77 of ASC memory is shown below. Decode the program segment represented by this object code.

```
1000  0000  0000  0000
0010  0000  1000  0001
0110  0000  0100  0101
0111  0000  0100  0101
0101  0000  0100  1101
1100  0001  1000  0010
1100  0010  1000  0111
0001  0010  1000  0011
0011  0000  1000  0000
0010  0000  1000  0000
1110  0010  0100  1011
1111  0001  0100  1101
0101  0000  0100  0111
0000  0000  0000  0000
```

4.12 Write ASC assembly language programs for the following. Start programs at locations #H0, using ORG 0 Statement.

a. Subtract an integer stored at memory location A from that at B and store the result at C.
b. Read several integers from the input device one at a time and store them in a memory location starting at Z. The input process should stop when the integer read has a value of 0.
c. Change the program in (b) above to store only positive integers at a location starting at POS.
d. Modify the program in (b) above to store the positive integers starting at POS and the negative integers starting at NEG.
e. Location #H50 contains an address pointing to the first entry in a table of integers. The table is also in the memory and the first entry is the number of entries in the table, excluding itself. Store the maximum and minimum valued integers at memory locations MAX and MIN, respectively.
f. SORT the entries in a table of *n* entries in increasing order of *magnitude*.
g. Multiply integers stored at memory locations A and B and store the result in C. Assume the product is small enough and can be represented in 16 bits. Note that multiplication is the repeated addition of multiplicand to itself multiplier times.
h. Compute the absolute value of each of the 50 integers located at the memory block starting at A and store them at the block starting at B.
i. Read a sequence of numbers and compute their minimum, maximum, and average values. Reading a value of 0 should terminate the reading process. What problems arise in computing the average?

4.13 Assemble each program in Problem 4.12 and list the object code in binary and hexadecimal forms.

4.14 What restrictions are to be imposed on the assembly language if a single-pass assembler is needed?

4.15 Numbers larger than $(2^{15} - 1)$ are required for certain applications. Two ASC words can be used to store each such number. What changes to the ASC instruction set are needed to enable addition of numbers that are each stored in two consecutive memory words?

4.16 SHR instruction must be enhanced to allow multiple shifts. The address field can be used to represent the shift count; for example,

```
SHR  5
```

implies a shift right by five bits. Discuss the assembler modifications needed to accommodate multiple-shift SHR. The hardware is capable of performing only one shift at a time.

# 5
# A Simple Computer: Hardware Design

The organization of a simple computer (ASC) provided in Chapter 4 is the programmer's view of the machine. We will illustrate the complete hardware design of ASC in this chapter. Assume that the design follows the sequence of eight steps listed here.

1. Selection of an instruction set
2. Word size selection
3. Selection of instruction and data formats
4. Register set and memory design
5. Data and instruction flowpath design
6. Arithmetic and logic unit design
7. Input/output mechanism design
8. Generation of control signals and design of control unit.

In practice, design of a digital computer is an iterative process. A decision made early in the design process may have to be altered to suit some parameter at a later step. For example, instruction and data formats may have to be changed to accommodate a better data or instruction flowpath design.

A computer architect selects an architecture for the machine based on cost and performance tradeoffs, and a computer designer implements the architecture using the hardware and software components available. The complete process of the development of a computer system thus can also be viewed as consisting of two phases, design and implementation. Once the architect derives an architecture (design phase), each subsystem in the architecture can be implemented in several ways depending on the available technology and requirements. We will not distinguish between these two phases in this chapter. We restrict this chapter to the design of hardware

components of ASC and use memory elements, registers, flip-flops, and logic gates as components in the design.

Architectural issues of concern at each stage in the design are described in subsequent chapters with reference to architectures of commercially available machines. We will now describe the program execution process in order to depict the utilization of each of the registers in ASC.

## 5.1 PROGRAM EXECUTION

Once the object code is loaded into the memory, it can be executed by initializing the program counter to the starting address and activating the START switch on the console. Instructions are then fetched from the memory and executed in sequence until an HLT instruction is reached or an error condition occurs. The execution of an instruction consists of two phases:

1. Instruction fetch
2. Instruction execute.

### 5.1.1 Instruction Fetch

During instruction fetch, the instruction word is transferred from the memory into the *instruction register* (IR). To accomplish this, the contents of the program counter (PC) are first transferred into the MAR, a memory read operation is performed to transfer the instruction into the MBR, and the instruction is then transferred to the IR. While memory is being read, the control unit uses its internal logic to add 1 to the contents of the program counter so that the program counter points to the memory word following the one from which the current instruction is fetched. This sequence of operations constitutes the fetch phase and is the same for all instructions.

### 5.1.2 Instruction Execution

Once the instruction is in the instruction register, the opcode is decoded and a sequence of operations is brought about to retrieve the operands (if needed) from the memory and to perform the processing called for by the opcode. This is the execution phase and is unique for each instruction in the

instruction set. For example, for the LDA instruction the effective address is first calculated, then the contents of the memory word at the effective address are read and transferred into ACC. At the end of the execute phase, the machine returns to the fetch phase.

ASC uses an additional phase to compute the effective address if the instruction uses the indirect addressing mode, since such computation involves reading an address from the memory. This phase is termed *defer phase*.

The fetch, defer, and execute phases together form the so called *instruction cycle*. Note that an instruction cycle need not contain all the three phases. The fetch and execute phases are required for all instructions, and the defer phase is required only if the indirect addressing is called for.

---

**Example 5.1** Figure 5.1 gives a sample program and shows the contents of all the ASC registers during the execution of the program.

---

## 5.2 DATA, INSTRUCTION, AND ADDRESS FLOW

A detailed analysis of the flow of data, instructions, and addresses during instruction fetch, defer, and execute phases for each instruction in the instruction set is required to determine the signal flow paths needed between registers and memory. Such an analysis for ASC, based on the execution process shown in Fig. 5.1, follows.

### 5.2.1 Fetch Phase

Assuming that the PC is loaded with the address of the first instruction in the program, the following set of register transfers and manipulations is needed during the fetch phase of each instruction:

| | |
|---|---|
| MAR ← PC. | |
| Read memory. | Instruction is transferred to MBR; i.e., MBR ← M[MAR]. |
| IR ← MBR | Transfer instruction to IR. |
| PC ← PC + 1 | Increment PC by 1. |

| Location (hexadecimal) | | Instruction | |
|---|---|---|---|
| | | ORG | 0 |
| 0000 | BEGIN | LDX | TWO,1 |
| 0001 | | LDA | ZERO |
| 0002 | | ADD | X |
| 0003 | LOOP | ADD | X,1 |
| 0004 | | TDX | LOOP,1 |
| 0005 | | STA* | Y |
| 0006 | | HLT | |
| 0007 | TWO | BSC | 2 |
| 0008 | ZERO | BSC | 0 |
| 0009 | X | BSC | 5 |
| 000A | | BSC | 7 |
| 000B | | BSC | 34 |
| 000C | Y | BSC | 76 |
| | | END | BEGIN |

(a) Source program

| Phase | Operations | Comments |
|---|---|---|
| INITIALIZE | PC ← 0 | |
| FETCH | MAR ← #H0000 | |
| | READ MEMORY | |
| | MBR ← Instruction LDX TWO, 1 | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0001 | Increment PC |
| EXECUTE | MAR ← TWO | Address of TWO to MAR |
| | READ MEMORY | |
| | MBR ← M[TWO] | |
| | Index1 ← MBR | Load 2 into Index Register 1 |
| FETCH | MAR ← #H0001 | |
| | READ MEMORY | |
| | MBR ← Instruction LDA ZERO | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0002 | Increment PC |
| EXECUTE | MAR ← ZERO | Address of ZERO to MAR |
| | READ MEMORY | |
| | MBR ← M[ZERO] | |
| | ACC ← MBR | Load 0 into ACC |

**Figure 5.1** Execution process

| Phase | Operations | Comments |
|---|---|---|
| FETCH | MAR ← #H0002 | |
| | READ MEMORY | |
| | MBR ← Instruction ADD X | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0003 | Increment PC |
| EXECUTE | MAR ← X | Address of X to MAR |
| | READ MEMORY | |
| | MBR ← M[X] | |
| | ACC ← MBR + ACC | Add 5 to ACC |
| FETCH | MAR ← #H0003 | |
| | READ MEMORY | |
| | MBR ← Instruction ADD X,1 | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0004 | Increment PC |
| EXECUTE | MAR ← X + (INDEX1) | X + 2 to MAR |
| | READ MEMORY | |
| | MBR ← M[X + 2] | |
| | ACC ← MBR + ACC | Add 34 to ACC |
| FETCH | MAR ← #H0004 | |
| | READ MEMORY | |
| | MBR ← Instruction TDX LOOP,1 | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0005 | Increment PC |
| EXECUTE | INDEX1 ← INDEX1 − 1 | Decrement Index1 |
| | PC ← #H0003 | Jump to LOOP since index1 is not 0 |
| FETCH | MAR ← #H0003 | |
| | READ MEMORY | |
| | MBR ← Instruction ADD X,1 | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0004 | Increment PC |
| EXECUTE | MAR ← X + (INDEX1) | Address X + 1 to MAR |
| | READ MEMORY | |
| | MBR ← M[X + 1] | |
| | ACC ← MBR + ACC | Add 7 to ACC |
| FETCH | MAR ← #H0004 | |
| | READ MEMORY | |
| | MBR ← Instruction TDX LOOP,1 | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0005 | Increment PC |

**Figure 5.1** (Continued) *(continues)*

| Phase | Operations | Comments |
|---|---|---|
| EXECUTE | INDEX1 ← INDEX1 − 1 | Decrement Index1, Index1 is 0, Do not jump to Loop |
| FETCH | MAR ← #H0005 | |
| | READ MEMORY | |
| | MBR ← Instruction STA* Y | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0006 | Increment PC |
| DEFER | MAR ← Y | Indirect Addressing |
| | READ MEMORY | |
| | MBR ← 76 | Effective Address in MBR |
| EXECUTE | MAR ← MBR | Address 76 to MAR |
| | MBR ← ACC | |
| | WRITE MEMORY | |
| | M[76] ← MBR | ACC stored at 76 |
| FETCH | MAR ← #H0006 | |
| | READ MEMORY | |
| | MBR ← Instruction HLT | |
| | IR ← MBR | Instruction in IR |
| | PC ← #H0007 | Increment PC |
| EXECUTE | Halt | Stops execution |

(b) Execution trace

**Figure 5.1** (Continued)

### 5.2.2 Address Calculations

For one-address instructions the following address computation capabilities are needed:

| | | |
|---|---|---|
| Direct: | MAR ← $IR_{7-0}$ | MAR ← IR(ADRS) |
| Indexed: | MAR ← $IR_{7-0}$ + INDEX. | (INDEX refers to the index register selected by the index flag.) |
| Indirect: | MAR ← $IR_{7-0}$ | Get the new address. |
| | READ Memory. | |
| | MAR ← MBR. | Address to MAR. |
| Preindexed-indirect | MAR ← $IR_{7-0}$ + INDEX. | |
| | READ Memory. | Get the new address. |
| | MAR ← MBR. | |

In the above, *effective address* is assumed to be in MAR at the end of address calculation, just before the execution of the instruction begins. Using the concatenation operator "¢", the transfer MAR $\leftarrow$ $IR_{7-0}$ should be further refined as

$$MAR \leftarrow 00000000 \text{ ¢ } IR_{7-0}$$

meaning that the most significant 8 bits of MAR receive 0s. The concatenation of 0s to the address field of IR as above is assumed whenever the contents of the IR address field are transferred to a 16-bit destination or the IR address field is added to an index register. The memory operations can also be designated as

READ memory MBR $\leftarrow$ M[MAR], and
WRITE memory M[MAR] $\leftarrow$ MBR.

Note that if the contents of the index register are 0,

$$MAR \leftarrow IR_{7-0} + INDEX$$

can be used for direct address computation also. This fact will be used later in this chapter.

Assuming that all arithmetic is performed by one arithmetic unit, the instruction and address flowpaths required for fetch cycle and address computation in ASC are as shown in Fig. 5.2. (For a list of mnemonic codes, see Table 4.1, page 193.)

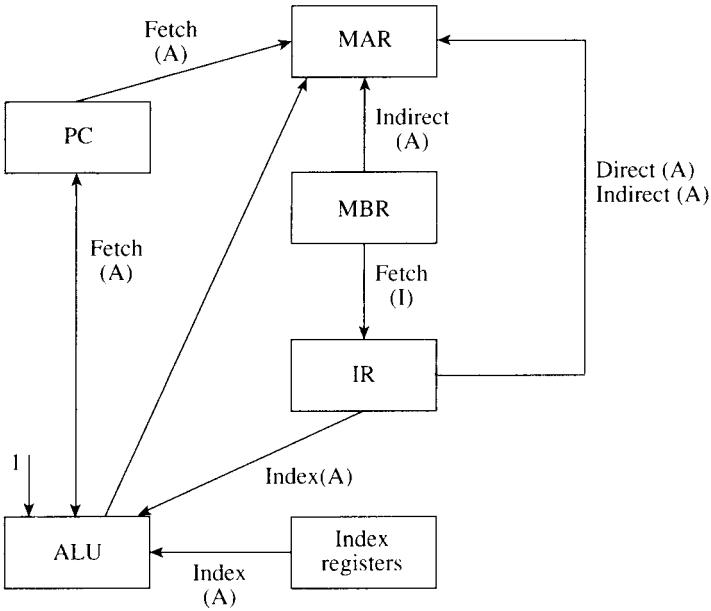
### 5.2.3 Execution Phase

The detailed data flow during the execution of each instruction is determined by the analysis shown in Table 5.1.

## 5.3 BUS STRUCTURE

The data and address transfers shown in Fig. 5.2 can be brought about either by a point-to-point interconnection of registers or by a bus structure connecting all the registers in ASC. Bus structures are commonly used because a point-to-point interconnection becomes complex as the number of registers to be connected increases.

The most common bus structures are *single bus* and *multibus*. In a single-bus structure, all data and address flow is through one bus. A multibus structure typically consists of several buses, each dedicated to certain transfers; for example, one bus could be a data bus and the other could be



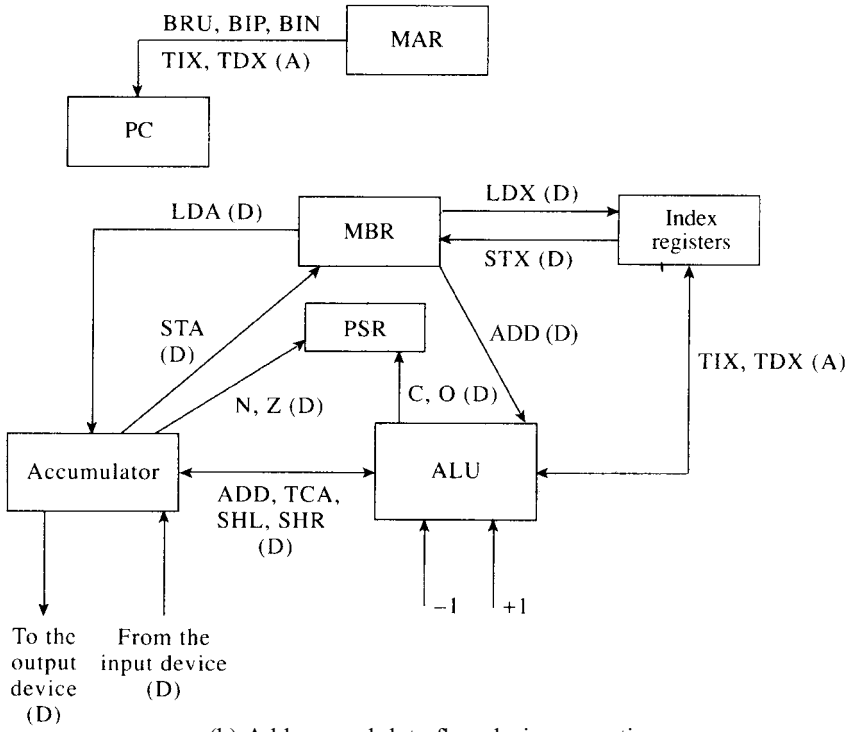

(a) Instruction and address flow during fetch phase and address computation phase

Note: A indicates address, I indicates instruction, and D indicates data.
All paths are 16 bits wide unless indicated otherwise.
Input 1 into the arithmetic unit is a constant input required to increment PC.
Constant inputs are required for incrementing and decrementing PC and index registers.
Refer to Table 4.1, ASC instruction set.

**Figure 5.2** Register transfer paths of a simple computer (ASC)

an address bus. Multibus structures provide the advantage of tailoring each bus to the set of transfers it is dedicated to and permits parallel operations. Single-bus structures have the advantage of uniformity of design. Other characteristics to be considered in evaluating bus structures are the amount of hardware required and the data transfer rates possible.

Figure 5.3 shows a bus structure for ASC. The bus structure is realized by recognizing that two operands are required for arithmetic operations using the arithmetic unit of ASC. Each operand will be on a bus (BUS1 and BUS2) feeding the arithmetic unit. The output of the arithmetic unit will be on another bus (BUS3). For transfers that do not involve any arithmetic operation, two direct paths are assumed through the arithmetic unit. These paths either transfer BUS1 to BUS3 or transfer BUS2 to BUS3. The contents of each bus during other typical operations are listed below:




(b) Address and data flow during execution

**Figure 5.2** (Continued)

| Operation | BUS1 | BUS2 | BUS3 |
|---|---|---|---|
| Increment PC | PC | 1 (constant) | PC + 1 |
| Indexing | 00000000 ¢ $IR_{7-0}$ | INDEX | Indexed address |
| ADD | ACC | MBR | ACC + MBR |

The operation of the bus structure shown in Fig. 5.3 is similar to that in Fig. 2.45. The data enters a register at the rising edge of the clock. The clock and all the other required control signals to select source and destination registers during a bus transfer are generated by the control unit. Figure 5.4 (a) shows the detailed bus transfer circuit for the ADD operation. The timing diagram is shown in (b). Control signals ACC to BUS1 and MBR to BUS2 select ACC and MBR as source registers for BUS1 and BUS2 respectively. The control signal ADD enables the add operation in the ALU. The signal BUS3 to ACC selects the ACC as the destination for BUS3. All these control signals are active at the same time and stay active long enough to

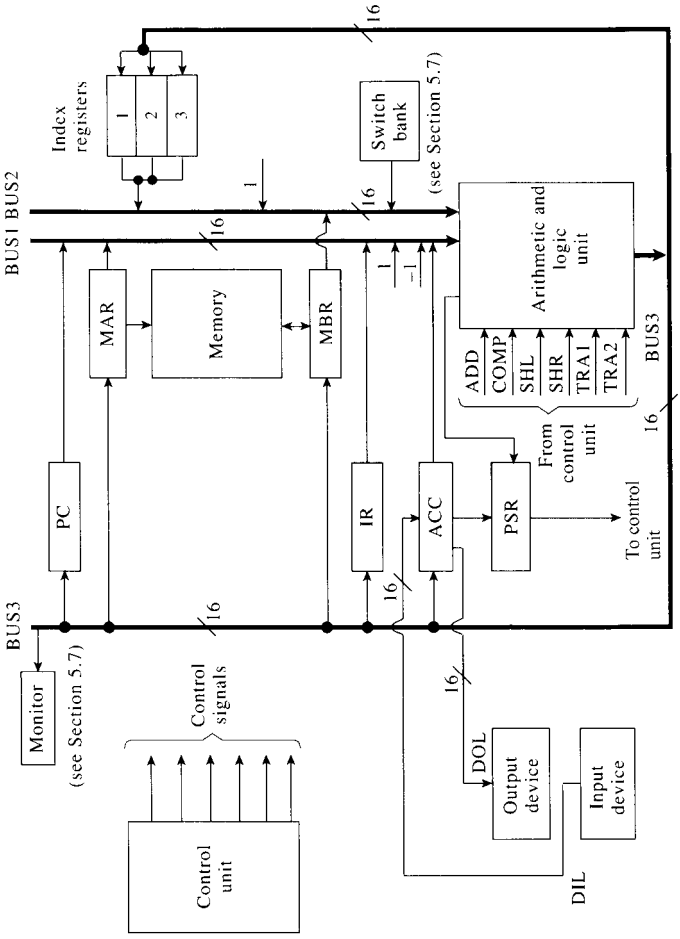
**Table 5.1** Analysis of data flow during execution

| Mnemonic | Operations | Comments |
|---|---|---|
| HLT | RUN ← 0. | Stop |
| LDA | READ memory. | Data into MBR. (The effective address of the operand is in MAR.) |
| | ACC ← MBR. | |
| STA | MBR ← ACC. | |
| | WRITE memory. | |
| ADD | READ memory. | Data into MBR. |
| | ACC ← ACC + MBR. | |
| TCA | ACC ← ACC.′ | Two-step computation of |
| | ACC ← ACC + 1. | the 2s complement. |
| BRU | PC ← MAR. | Address goes to PC. |
| BIP | IF $ACC_{15} = 0$ and $ACC_{14-0} \neq 0$ THEN PC ← MAR. | |
| BIN | If $ACC_{15} \neq 0$ THEN PC ← MAR. | |
| RWD | ACC ← Input. | Input device data buffer contents to ACC. |
| WWD | Output ← ACC. | ACC to output device. |
| SHL | ACC ← $ACC_{14-0}$ ¢ 0. | Zero fill. |
| SHR | $ACC_{14-0}$ ← $ACC_{15-1}$ | $ACC_{15}$ not altered. |
| LDX | READ memory | |
| | INDEX ← MBR. | |
| STX | MBR ← INDEX. | |
| | WRITE memory. | |
| TIX | INDEX ← INDEX + 1. | |
| | IF INDEX = 0 THEN PC ← MAR. | |
| TDX | INDEX ← INDEX – 1. | |
| | IF INDEX ≠ 0 THEN PC ← MAR. | |

Note: The data paths required for the operations given in Table 5.1 are shown in Figure 5.2. All paths are 16 bits wide unless indicated otherwise. Also note that indexing is not allowed in index-register reference instructions LDX, STX, TIX, and TDX.

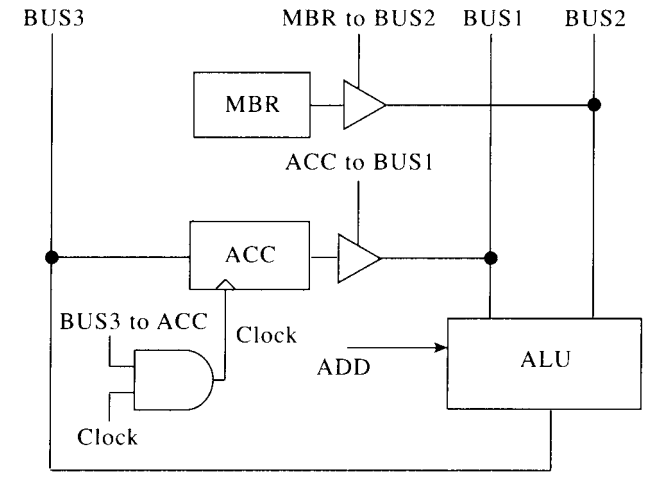
complete the bus transfer. The data enters the ACC at the rising edge of the clock pulse, after which the control signals become inactive.

The time required to transfer a data unit from the source to the destination through the ALU is the *register transfer* time. The clock frequency must be such that the slowest register transfer is accomplished in a clock period. (In ASC, the slowest transfer is the one that involves an ADD operation.) Thus, the register transfer time dictates the speed of transfer on the bus and hence the processing speed (*cycle time*) of the processor.

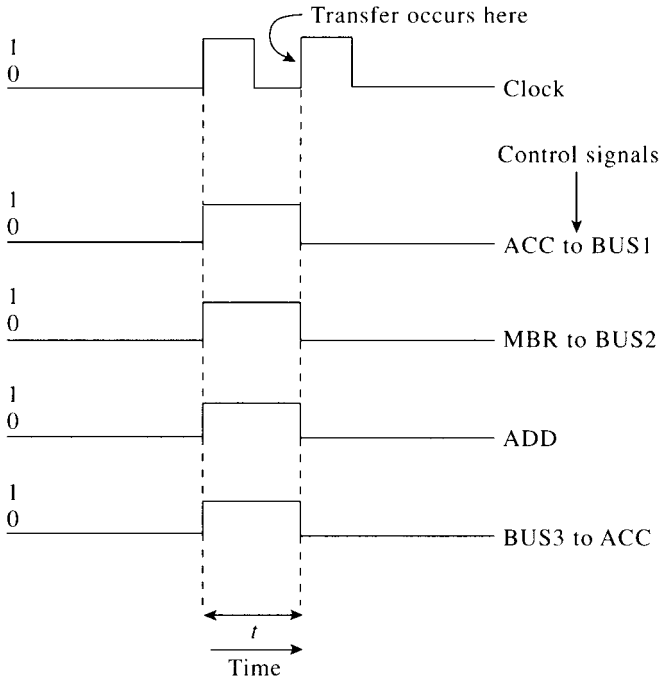



Notes: Each bit of PC and ACC is a master–slave flip-flop.
1 and –1 represent constant registers connected to the respective bus; they are 16-bit constants represented in 2s complement form.
Only the control signals to ALU are shown; other control signals are not shown.

**Figure 5.3** ASC bus structure




(a) Circuit




(b) Timing

**Figure 5.4** Timing of ADD operation on the bus structure

Note also that the contents of the ACC and MBR cannot be altered until their sum enters the ACC, since the ALU is a combinational circuit. In order to accommodate this feedback operation on the ACC, it must be configured using the master/slave flip-flops. Similarly, to accommodate the increment PC operation, the PC must also be configured using master/slave flip-flops.

Input and output transfers are performed on two separate 16-bit paths: data input lines (DIL) and data output lines (DOL) connected to and from the accumulator. This input/output scheme was selected for simplicity. Alternatively, DIL and DOL could have been connected to one of the three buses.

A single bus structure is possible for ASC. In this structure, either one of the operands of the two operand operations or the result must be stored in a buffer register before it can be transmitted to the destination register. Thus, there will be some additional transfers in the single-bus structure and some operations will take longer to complete, thereby making the structure slower than the multibus structure.

Transfer of data, instructions, and addresses on the bus structure is controlled by a set of control signals generated by the control unit of the machine. Detailed design of the control unit is illustrated in Section 5.6.

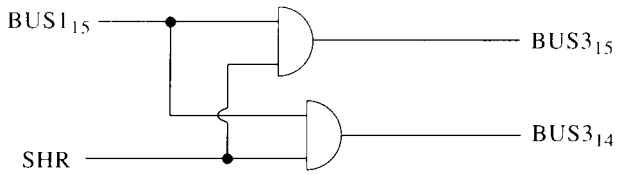
## 5.4 ARITHMETIC AND LOGIC UNIT

The arithmetic and logic unit (ALU) of ASC is the hardware which performs all arithmetic and logical operations. The instruction set implies that the ALU of ASC must perform addition of two numbers, compute the 2s complement of a number, and shift the contents of the accumulator either right or left by one bit. Additionally, the ASC ALU must directly transfer either of its inputs to its output to support data transfer operations such as IR $\leftarrow$ MBR and MAR $\leftarrow$ IR.

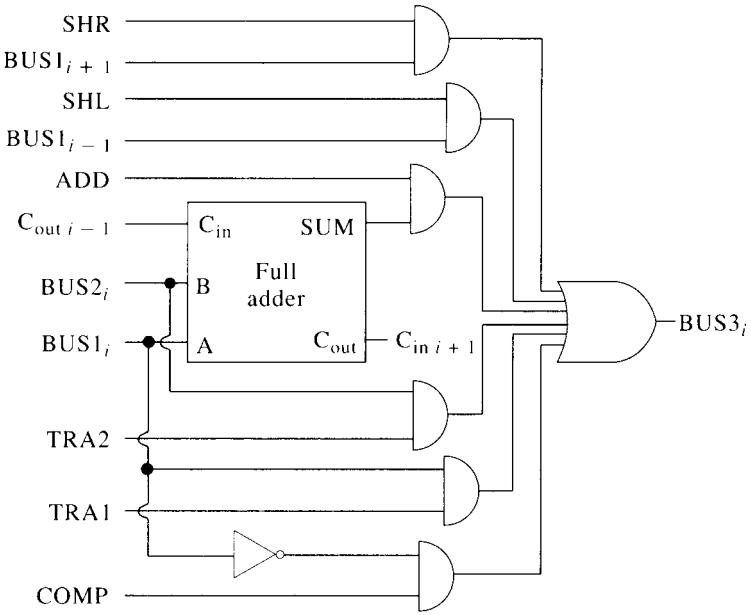
We will assume that the control unit of the machine provides the appropriate control signals to enable the ALU to perform one of these operations. Since BUS1 and BUS2 are the inputs and BUS3 is the output of ALU, the following operations must be performed by the ALU:

ADD: BUS3 $\leftarrow$ BUS1 + BUS2.
COMP: BUS3 $\leftarrow$ BUS1.$'$
SHR: BUS3 $\leftarrow$ $\text{BUS1}_{15}$ ¢ $\text{BUS1}_{15-1}$.
SHL: BUS3 $\leftarrow$ $\text{BUS1}_{14-0}$ ¢ 0.
TRA1: BUS3 $\leftarrow$ BUS1.
TRA2: BUS3 $\leftarrow$ BUS2.

ADD, COMP, SHR, SHL, TRA1, and TRA2 are the control signals generated by the control unit, and bit positions of the buses are numbered 15 through 0, left to right. Each of the control signals activates a particular operation in ALU. Only one of these control signals may be active at any time. Figure 5.5 shows a typical bit of ALU and its connections to the bits of BUS1, BUS2, and BUS3. A functional description follows of the ALU corresponding to the above control signals.




(a) Sign-bit circuit




(b) Typical bit

Note: During SHL, all lines connected to BUS3$_0$ are 0, and hence the LSB is automatically zero filled; no special circuitry is needed. During SHR, BUS1$_0$ is not connected to any bit of BUS3 and hence is lost.

**Figure 5.5** Logic diagram of ASC ALU

*ADD*: The addition circuitry consists of fifteen full adders and one half-adder for the least significant bit (bit 0). The sum output of each adder is gated through an AND gate with the ADD control signal. The carry output of each adder is input to the carry-in of the adder for the next most significant bit. The half-adder for bit 0 has no carry-in and the carry-out from bit 15 is the CARRY flag bit in the PSR. The accumulator contents on BUS1 are added to the contents of MBR on BUS2, and the result is stored in the accumulator.

*COMP*: The complement circuitry consists of sixteen NOT gates, one for each bit on BUS1. Thus, the circuitry produces the 1s complement of a number. The output of each NOT gate is gated through an AND gate with the COMP control signal. The operand for complement is the contents of the accumulator and the result is stored in the accumulator. TCA (2s complement accumulator) command is accomplished by taking the 1s complement of an operand first and then adding 1 to the result.

*SHR*: For shifting a bit pattern right, each bit of BUS1 is routed to the next least significant bit of BUS3. This transfer is gated by SHR control signal. The least significant bit of BUS1 ($BUS1_0$) is lost in the shifting process, while the most significant bit of BUS1 ($BUS1_{15}$) is routed to both the most significant bit ($BUS3_{15}$) and the next least significant bit ($BUS3_{14}$) of BUS3. Thus, the left-most bit of the output is "sign" filled.

*SHL*: For shifting a bit pattern left, each bit of BUS1 is routed to the next most significant bit of BUS3. SHL control signal is used to gate this transfer. The most significant bit on BUS1 ($BUS1_{15}$) is lost in the shifting process, while the least significant bit of BUS3 ($BUS3_0$) is zero filled.

*TRA1*: This operation transfers each bit from BUS1 to the corresponding bit on BUS3, each bit gated by TRA1 signal.

*TRA2*: This operation transfers each bit of BUS2 to the corresponding bit of BUS3, each bit gated by TRA2 control signal.

The carry, negative, zero and overflow bits of the PSR are set or reset by the ALU, based on the content of the ACC at the end of each operation involving the ACC. Figure 5.6 shows the circuits needed. The carry bit is simply the $C_{out}$ from bit position 15. $BUS3_{15}$ forms the negative bit. Zero bit is set only if all the bits of BUS3 are zero. Overflow occurs if the sum exceeds $(2^{15} - 1)$ during addition. This can be detected by observing the sign bits of the operands and the result. Overflow occurs if the sign bits of the operands are each 1 and the sign bit of the result is 0, or vice versa. Also, during SHL if the sign bit changes, an overflow results. This is detected by comparing $BUS1_{14}$ with $BUS1_{15}$. Note that the PSR bits are updated simultaneously with the updating of the ACC contents with the results of the operation. Figure 5.6 also shows the circuit to derive the accumulator-positive condition (i.e., ACC is neither negative nor zero) required during the BIP instruc-

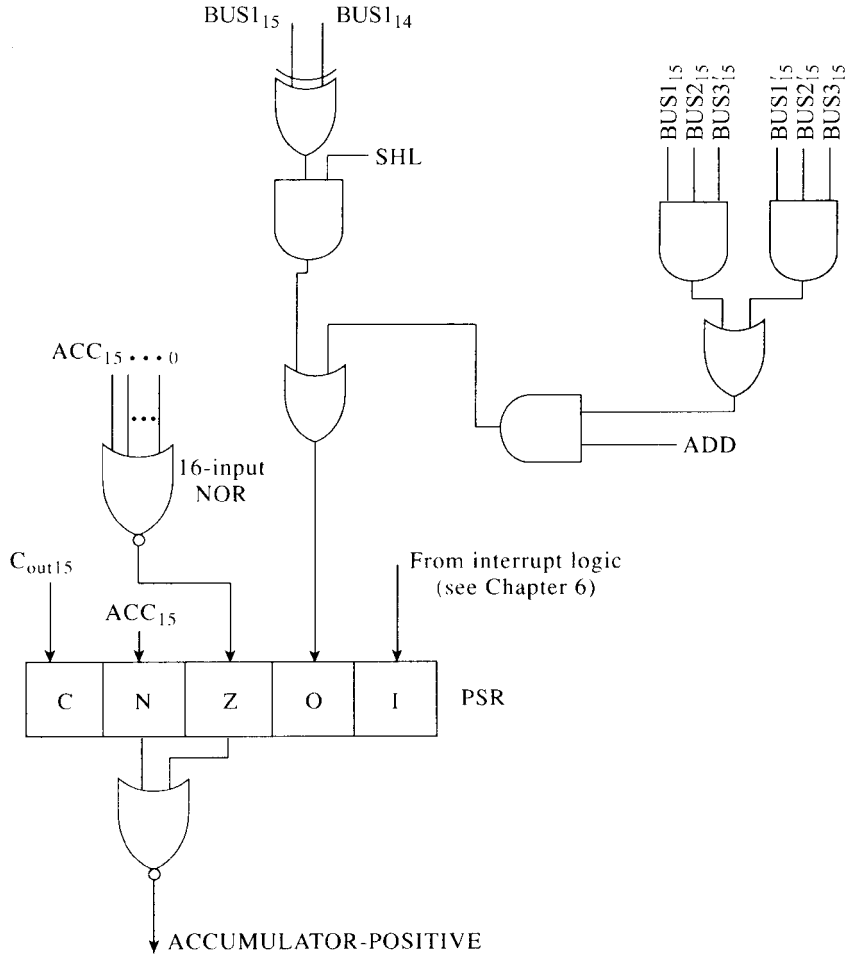



**Figure 5.6** Status generation circuits

tion execution. The interrupt bit of the PSR is set and reset by the interrupt logic, discussed in Chapter 6.

## 5.5 INPUT/OUTPUT

ASC is assumed to have one input device and one output device. Both these functions could very well be performed by a terminal with a keyboard for input and a display or printer for the output. We assume that the input and

output devices transfer a 16-bit data word into or out of the ACC, respectively.

We will base our design on the simplest input/output (I/O) scheme, called *programmed I/O*.

The ALU and the control unit together form the central processing unit (CPU), which we will also refer to as the processor. In the programmed I/O scheme, during the execution of the RWD instruction, the CPU commands the input device to send a data word and then waits. The input device gathers the data from the input medium and when the data is ready in its data buffer, informs the CPU that the data is ready. The CPU then gates the data into the ACC over the DILs. During the WWD, the CPU gates the ACC content onto DOLs, commands the outpt device to accept the data, and waits. When the data are gated into its data buffer, the output device informs the CPU of the data acceptance. The CPU then proceeds to execute the next instruction in the sequence.

The sequence of operations described above is known as the data communication *protocol* (or "handshake") between the CPU and the peripheral device. A DATA flip-flop in the control unit is used to facilitate the I/O handshake. The RWD and WWD protocols are described below in detail:

1. RWD
   a. CPU resets the DATA flip-flop.
   b. CPU sends a 1 on the INPUT control line, thus commanding the input device to send the data.
   c. CPU waits for the DATA flip-flop to be set by the input device.
   d. The input device gathers the data into its data buffer, gates it onto the DILs, and sets the DATA flip-flop.
   e. CPU now gates the DILs into the ACC, resets the INPUT control line, and resumes instruction execution.
2. WWD
   a. CPU resets the DATA flip-flop.
   b. CPU gates the ACC onto DOLs and sends a 1 on the OUTPUT control line, thus commanding the output device to accept the data.
   c. CPU waits for the DATA flip-flop to be set by the output device.
   d. The output device, when ready, gates DOL into its data buffer and sets the DATA flip-flop.
   e. CPU resets the OUTPUT control line, removes data from the DOLs, and resumes instruction execution.

As can be seen from the preceding protocols, the CPU controls the complete I/O process and waits for the input and output to occur. Since the I/O devices are much slower than the CPU, and the CPU idles waiting for the data, the programmed I/O scheme is the slowest of the I/O schemes used in practice. But it is simple to design and generally has a low overhead in terms of the I/O protocols needed, especially when small amounts of data are to be transferred. Also note that this scheme is adequate in the environments in which the CPU cannot be allocated to any other task while the data transfer is taking place. Chapter 6 provides the details of the other popular I/O schemes and generalizes the ASC I/O scheme described in this chapter.

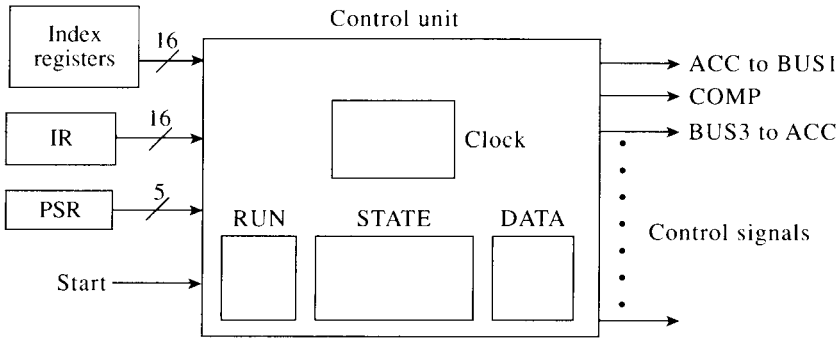
## 5.6 CONTROL UNIT

The control unit is the most complex block of computer hardware from a designer's point of view. Its function is to generate control signals needed by other blocks of the machine in a predetermined sequence to bring about the sequence of actions called for by each instruction.

Figure 5.7 shows a block diagram of the ASC control unit and lists all the external and internal inputs and the outputs (i.e., control signals produced.)

Inputs to the control units are:

1. The opcode, indirect bit, and index flag from IR.
2. Contents of the PSR.




(a) Block diagram

**Figure 5.7** Control unit

External inputs:

From registers

PSR contents
IR contents
Index registers contents

From console

SWITCH BANK contents
START
MASTER CLEAR

From I/O devices

Set DATA flip-flop

Internal Inputs: Contents of RUN and DATA flip-flops and STATE register
CLOCK

Signals generated for internal use:

State change signals

Fetch
Defer (indirect address)
Execute
Reset DATA flip-flop
Reset RUN flip-flop

Outputs:

| To memory | To ALU | To I/O |
|---|---|---|
| READ | TRA1 | INPUT |
| WRITE | TRA2 | OUTPUT |
| | ADD | |
| | COMP | |
| | SHR | |
| | SHL | |

To the bus structure

| | | | |
|---|---|---|---|
| ACC to BUS1 | INDEX to BUS2 | BUS3 to ACC | DIL to ACC |
| MAR to BUS1 | MBR to BUS2 | BUS3 to INDEX | ACC to DOL |
| $IR_{7-0}$ to BUS1 | 1 to BUS2 | BUS3 to MAR | |
| PC to BUS1 | SWITCHBANK to BUS2 | BUS3 to MBR | |
| 1 to BUS1 | | BUS3 to PC | |
| −1 to BUS1 | | BUS3 to MONITOR | |
| | | BUS3 to IR | |

(b) Signals

Note: CLOCK is connected to all the registers. SWITCH BANK, MONITOR, MASTER, CLEAR, START are console facilities

**Figure 5.7** (Continued)

3. Index register bits 0–15, to test for the zero or nonzero index register in TIX and TDX instructions.

In addition to these inputs, control signals generated by the control unit are functions of the contents of the following:

1. DATA flip-flop: Used to facilitate the handshake between the CPU and I/O devices.
2. RUN flip-flop: Set by the START switch on the console (see Section 5.7), indicates the RUN state of the machine. RUN flip-flop must be set for control signals to activate a microoperation. RUN flip-flop is reset by the HLT instruction.
3. STATE register: A 2-bit register used to distinguish between the three phases (states) of the instruction cycle. The control unit is thus viewed as a three-state sequential circuit.

### 5.6.1 Types of Control Units

As mentioned earlier, the function of the control unit is to generate the control signals in the appropriate sequence to bring about the instruction cycle that corresponds to each instruction in the program. In ASC an instruction cycle consists of three phases. Each phase in the instruction cycle is composed of a sequence of *microoperations*. A microoperation is one of the following:

1. A simple *register transfer* operation: the transfer of contents of one register to the other register.
2. A complex register transfer involving ALU, such as the transfer of the complement of the contents of a register, the sum of the contents of two registers, etc. to the destination register.
3. A memory read or write operation.

Thus, a machine instruction is composed of a sequence of microoperations (i.e., a *register transfer sequence*).

The CU of Fig. 5.7 can be implemented in two popular schemes:

1. *Hardwired control unit* (HCU). The outputs (control signals) of the CU are generated by the logic circuitry built of gates and flip-flops.
2. *Microprogrammed control unit (MCU)*. The sequence of microoperations corresponding to each machine instruction are stored in a read-only memory called *control ROM* (CROM). The sequence of microoperations is called the *microprogram*, and the microprogram consists of *microinstructions*. A microinstruc-

tion corresponds to one or more microoperations, depending on the CROM storage format.

In the following discussion, we will use the terms *register transfer* and *microoperation* interchangeably.

The MCU scheme is more flexible than the HCU scheme because in it the meaning of an instruction can be changed by changing the microinstruction sequence corresponding to that instruction, and the instruction set can be extended simply by including a new ROM containing the corresponding microoperation sequences. Hardware changes to the control unit thus are minimal in this implementation. In an HCU, any such change to the instruction set requires substantial changes to the hardwired logic. HCUs, however, are generally faster than MCUs and are used where the control unit must be fast. Most of the more recent machines have microprogrammed control units.

Among the machines that have an MCU, the degree to which the microprogram can be changed by the user varies from machine to machine. Some do not allow the user to change the microprogram, some allow partial changes and additions (for example, machines with *writable control store*), and some do not have an instruction set of their own and allow the user to microprogram the complete instruction set suitable for his application. This latter type of machine is called a *soft machine*. We will design an HCU for ASC in this section. An MCU design is provided in Section 5.8.

### 5.6.2 Hardwired Control Unit for ASC

A hardwired CU can either be synchronous or asynchronous. In a synchronous CU, each operation is controlled by a clock and the control-unit state can be easily determined knowing the state of the clock. In an asynchronous CU, completion of one operation triggers the next and hence no clock exists. Because of its nature, the design of an asynchronous CU is complex, but if it is designed properly it can be made faster than a synchronous CU. In a synchronous CU, the clock frequency must be such that the time between two clock pulses is sufficient to allow the completion of the slowest microoperation. This characteristic makes a synchronous CU relatively slow. We will design a synchronous CU for ASC.

### 5.6.3 Memory versus Processor Speed

The memory hardware is usually slower than the CPU hardware, although the speed gap is narrowing with the advances in hardware technology. Some memory organizations that help reduce this speed gap are described in

Chapter 8. We will assume a semiconductor RAM for ASC with an access time equal to two register transfer times. Thus, during a memory read, if the address is gated into the MAR along with the READ control signal, the data will be available in the MBR by the end of the next register transfer time. Similarly, if the data and address are provided in MBR and MAR respectively, along with the WRITE control signal, the memory completes writing the data by the end of the second register transfer time. These characteristics are shown in Fig. 5.8. Note that the contents of MAR cannot be altered until the read or write operation is completed.

### 5.6.4 Machine Cycles

In a synchronous CU, the time between two clock pulses (*register transfer time*) is determined by the slowest register transfer operation. In the case of ASC, the slowest register transfer is the one that involves the adder in the ALU. The register transfer time of a processor is known as the *processor cycle time*, or *minor cycle*. A *major cycle* of a processor consists of several minor cycles. Major cycles with either fixed or variable number of minor cycles have been used in practical machines. An instruction cycle typically consumes one or more major cycles.

To determine how long a major cycle needs to be, we will examine the fetch, address calculation, and execute phases in detail. The microoperations required during fetch phase can be allocated to minor cycles:

| Minor cycle | Microoperations | Comments |
|---|---|---|
| $T_1$: | MAR $\leftarrow$ PC, READ memory | |
| $T_2$: | PC $\leftarrow$ PC + 1 | Memory read time |
| $T_3$: | IR $\leftarrow$ MBR | |

Once the READ memory signal is issued at $T_1$, the instruction to be fetched will be available at the end of $T_2$, because the memory read operation requires two minor cycles. MAR cannot be altered during $T_2$, but the bus structure can be used during this time to perform other operations. Thus, we can increment the PC during this time slot, as required by the fetch phase. By the end of $T_3$, the instruction is available in IR.

The opcode is decoded by two 4-to-16 decoders (with active-high outputs) connected to the IR, as shown in Fig. 5.9. We will make our HCU design specific to the 16 instructions in the instruction set. Therefore, only one of the decoders is utilized in our design. When additional instructions are added, changes may be needed to more than one part of the HCU.

If the instruction in IR is a zero-address instruction, we can proceed to the execute cycle. If not, effective address must be computed. To facilitate

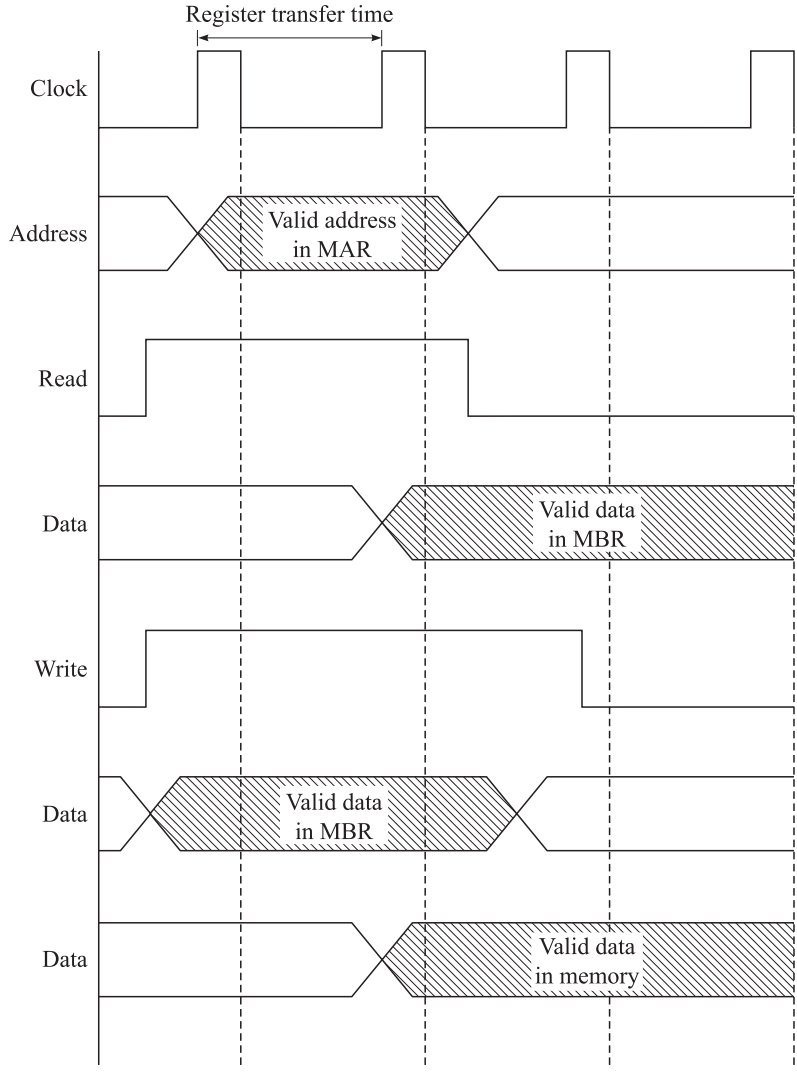



**Figure 5.8** ASC memory timing characteristics

the address computation, a 2-to-4 decoder is connected to the index flag bits of the IR, as shown in Fig. 5.9. The outputs of the decoder are used to connect the referenced index register to BUS2. Note that when the index flag is 00 (corresponding to no index register referencing), none of the three index registers are connected to BUS2; hence, the bus lines are all at zero.

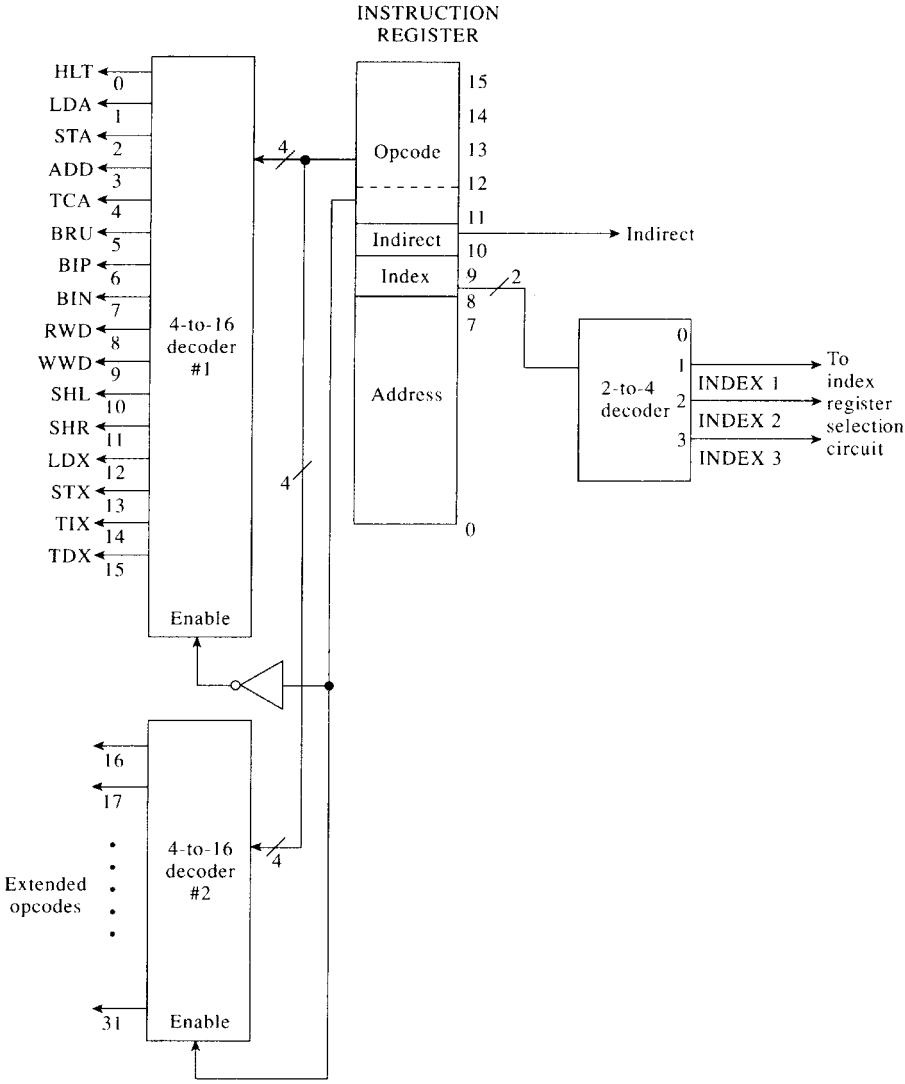



**Figure 5.9** IR Decoders

Fig. 5.10 shows the circuits needed to direct the data into the selected index register from BUS3 and the generation of ZERO-INDEX signal, which, if 0, indicates that the selected index register contains zero, as required during TIX and TDX instructions. In the following, INDEX refers to the index register selected by the circuit of Fig. 5.10.

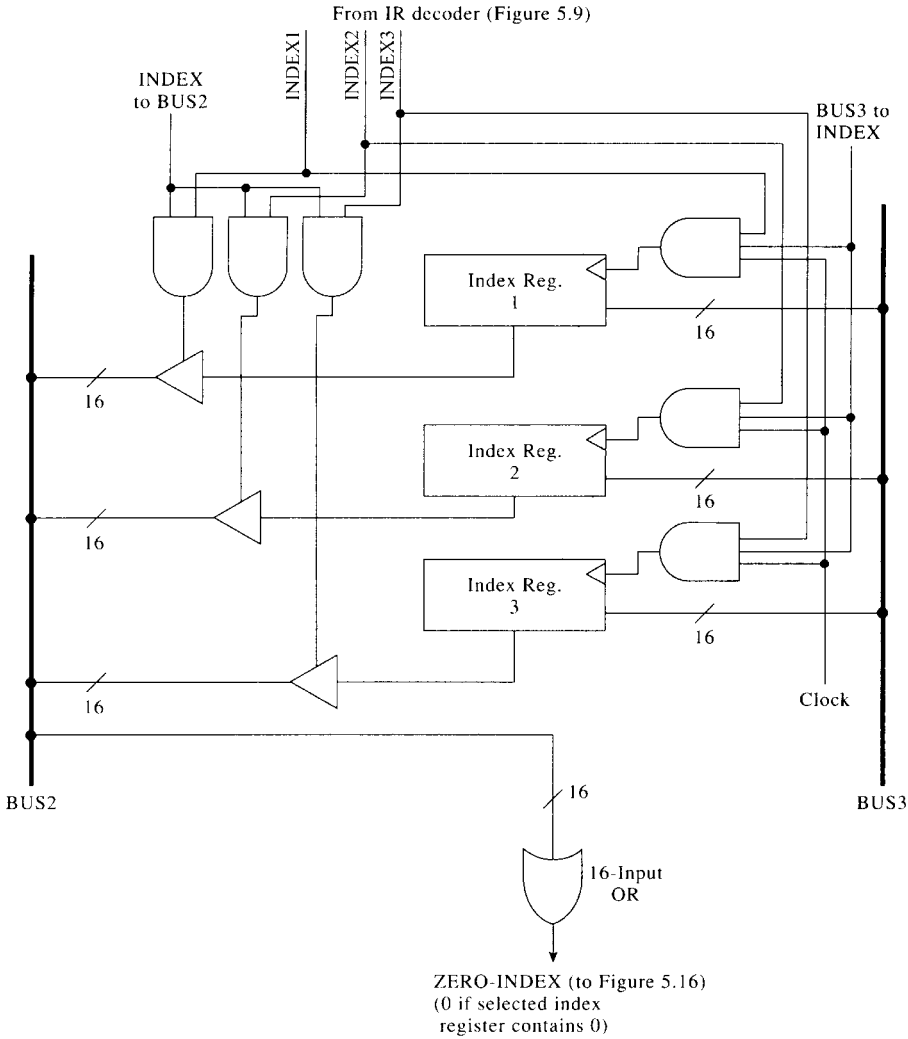



**Figure 5.10** Index register selection circuit

Thus, indexing can be performed during $T_4$:

$T_4$: MAR $\leftarrow$ 00000000 ç $IR_{7-0}$ + INDEX

where ç is the concatenation operator.

Indexing in $T_4$ is not needed for zero-address instructions. If the assembler inserts 0s into the unused fields of zero-address instructions, the

indexing operation during $T_4$ will not affect the execution of those instructions in any way. Also note that indexing is not allowed in LDX, STX, TIX, and TDX instructions. Microoperations in $T_4$ thus need to be altered for these instructions. From the opcode assignment it is seen that indexing is not required for instructions with an opcode in the range of 10000 through 11110. Hence, microoperations in $T_4$ can use the MSB of opcode $IR_{15}$ to inhibit indexing for index reference instructions. Thus,

$$T_4\text{: IF } IR_{15} = 0 \text{ THEN MAR} \leftarrow IR_{7-0} + \text{INDEX}$$
$$\text{ELSE MAR} \leftarrow IR_{7-0}.$$

The fetch phase thus consists of four minor cycles and accomplishes direct and indexed (if called for) address computations. Effective address is in the MAR at the end of the fetch phase. The first three machine cycles of the fetch phase are same for all 16 instructions. The fourth minor cycle will differ since the zero-address instructions do not need an address computation.

When indirect address is called for, the machine enters the defer phase, where a memory read is initiated and the result of the read operation is transferred into MAR as the effective address. We will thus redefine the functions of the three phases of ASC instruction cycle as follows:

FETCH: Includes direct and indexed address calculation.
DEFER: Entered only when indirect addressing is called for.
EXECUTE: Unique to each instruction.

Since the address (indexed if needed) is in MAR at the end of fetch phase, the defer phase can use the following time allocation:

$T_1$: READ
$T_2$: Wait.
$T_3$: MAR $\leftarrow$ MBR.
$T_4$ No operation.

Although only three minor cycles are needed for defer, we have made the defer phase four minor cycles long for simplicity since the fetch phase was four minor cycles long. This assumption results in some inefficiency. If $T_4$ of defer phase can be used to perform the microoperations needed during the execute phase of the instruction, the control unit would be more efficient, but its complexity would increase.

The execute phase differs for each instruction. Assuming a major cycle with four minor cycles again, the LDA instruction requires the following time allocation:

$T_1$: READ
$T_2$: Wait.
$T_3$: ACC ← MBR.
$T_4$: No operation.

In ASC, if the execution of an instruction cannot be completed in one major cycle, additional major cycles must be allocated for that instruction. This simplifies the design of HCU, but results in some inefficiency if all the minor cycles of the additional major cycle are not utilized by the instruction. (The alternative would have been to make the major cycles of variable length, which would have complicated the design.)

We will design the HCU as a synchronous circuit with three states corresponding to the three phases of the instruction cycle.

We will now analyze the microoperations needed by each ASC instruction and allocate them to whatever number of machine (major) cycles they require. We will maintain the machine cycles of constant length of four minor cycles for simplicity. For zero-address instructions where no address calculation is required, we will use the four minor cycles of the fetch state to perform execution phase microoperations, if possible.

ASC instruction cycle thus consists of one machine cycle for those zero-address instructions in which there is enough time left in the fetch machine cycle to complete the execution of the instruction; two machine cycles for some zero-address and single-address instructions with no indirect addressing; three cycles for single-address instructions with indirect addressing; and multiple machine cycles (depending on I/O wait time) for I/O instructions.

Table 5.2 lists the complete set of microoperations for LDA instruction and the control signals needed to activate those microoperations. The set of control signals to activate a microoperation must be generated simultaneously by the control unit. Note that the microoperations (and control signals) that are simultaneous are separated by a comma (,). A period (.) indicates the end of such a set of operations (signals). The conditional microoperations (signals) are represented using the notation:

IF condition THEN operation(s) ELSE operation(s).

The ELSE clause is optional, if an alternate set of operations is not required when the "condition" is not true (or 1).

The transitions between fetch (F), defer (D), and execute (E) states are allowed only in minor cycle 4 ($CP_4$) of each machine cycle, to retain the simplicity of design. If a complete machine cycle is not needed for a particular set of operations, these state transitions could occur earlier in the machine cycle, but the state transition circuit would be more complex.

**Table 5.2** Microoperations for LDA

| Machine cycle | Minor cycle (clock pulse) | Microoperations | Control signals |
|---|---|---|---|
| FETCH | $CP_1$ | MAR ← OC,<br>READ MEMORY | PC to BUS1, TRA1, BUS3 to MAR, READ. |
| | $CP_2$ | PC ← PC + 1. | PC to BUS1, 1 to BUS2, ADD, BUS3 to PC. |
| | $CP_3$ | IR ← MBR. | MBR to BUS2, TRA2, BUS3 to IR. |
| | $CP_4$ | IF $IR_{15} = 0$ THEN<br>MAR ← $IR_{7-0}$+ INDEX<br>ELSE<br>MAR ← $IR_{7-0}$,<br>IF $IR_{10} = 1$ THEN STATE ← D<br>ELSE STATE ← E. | IF $IR_{15} = 0$, THEN $IR_{7-0}$ to BUS1, INDEX to BUS2, ADD, BUS3 to MAR.<br>ELSE<br>$IR_{7-0}$ to BUS1, TRA1, BUS3 to MAR,<br>IF $IR_{10} = 1$, THEN D to STATE<br>ELSE E to STATE. |
| DEFER | $CP_1$ | READ MEMORY. | READ. |
| | $CP_2$ | WAIT. | – |
| | $CP_3$ | MAR ← MBR. | MBR to BUS2, TRA2, BUS3 to MAR. |
| | $CP_4$ | STATE ← E. | E to STATE. |
| EXECUTE | $CP_1$ | READ MEMORY. | READ. |
| | $CP_2$ | WAIT. | – |
| | $CP_3$ | ACC ← MBR. | MBR to BUS2, TRA2, BUS3 to ACC. |
| | $CP_4$ | STATE ← F. | F to STATE. |

The state transitions are represented by the operations STATE ← E, STATE ← D, and STATE ← F. These are equivalent to transferring the codes corresponding to each state into the STATE register. We will use the following coding:

| Code | State |
|---|---|
| 00 | F |
| 01 | D |
| 10 | E |
| 11 | Not used |

Note that in $CP_2$ of the fetch cycle, a constant register containing a 1 is needed to facilitate increment PC operation. Such a constant register is connected to BUS2 by the signal 1 to BUS2. In $CP_4$ of the fetch cycle, indexing is controlled by $IR_{15}$ and indirect address computation is controlled by $IR_{10}$. The state transition is either to defer (D) or to execute (E), depending on whether $IR_{10}$ is 1 or 0, respectively.

We will now analyze the remaining instructions to derive the complete set of control signals required for ASC.

### 5.6.5 One-address Instructions

Fetch and defer states are identical to the ones shown in Table 5.2 for all one-address instructions. Table 5.3 lists the execute-phase microprograms for these instructions.

### 5.6.6 Zero-Address Instructions

The microoperations during the first three minor cycles of fetch cycle will be similar to that of LDA, for zero-address instructions also. Since there is no address computation, some of the execution-cycle operations can be performed during the fourth minor cycle and no execution cycle is needed. Microoperations for all zero-address instructions are listed in Table 5.4. TCA is performed in two steps and hence needs the execution cycle. SHR, SHL, and HLT are completed in the fetch cycle itself.

**Table 5.3** Microoperations for One-address Instruction

| Machine cycle | Minor cycle | STA | ADD | BRU | LDX | STX |
|---|---|---|---|---|---|---|
| EXECUTE | $CP_1$ | – | READ. | PC ← MAR. | READ. | – |
| | $CP_2$ | – | – | – | – | – |
| | $CP_3$ | MBR ← ACC, WRITE. | ACC ← MBR + ACC. | – | INDEX ← MBR. | MBR ← INDEX, WRITE |
| | $CP_4$ | STATE ← F. | STATE ← F. | STATE ← F. | STATE ← F. | STATE ← F. |

| Machine cycle | Minor cycle or clock pulse | TIX | TDX | BIP | BIN |
|---|---|---|---|---|---|
| EXECUTE | $CP_1$ | INDEX ← INDEX + 1. | INDEX ← INDEX – 1. | – | – |
| | $CP_2$ | IF ZERO-INDEX = 0 THEN PC ← MAR. | IF ZERO-INDEX ≠ 0 THEN PC ← MAR. | – | – |
| | $CP_3$ | – | – | – | – |
| | $CP_4$ | STATE ← F. | STATE ← F. | IF ACCUMULATOR – POSITIVE = 1 THEN PC ← MAR · STATE ← F. | IF N = 1 THEN PC ← MAR · STATE ← F. |

**Table 5.4** Microoperations for Zero-address Instructions

| Cycle | Minor cycle | RWD | WWD | TCA | SHR | SHL | HLT |
|---|---|---|---|---|---|---|---|
| FETCH | $CP_1$ | | | | | | |
| | $CP_2$ | Same microoperations as those for LDA. | | | | | |
| | $CP_3$ | | | | | | |
| | $CP_4$ | DATA ← 0,<br>INPUT ← 1,<br>STATE ← E. | DATA ← 0,<br>OUTPUT ← 1,<br>STATE ← E,<br>DOL ← ACC. | STATE ← E. | ACC ← SHR(ACC). | ACC ← SHL (ACC). | RUN ← 0 |
| EXECUTE | $CP_1$ | – | – | AC ← ACC′. | | | |
| | $CP_2$ | – | – | ACC ← ACC + 1. | Execution cycle not needed; remains in F state. | | |
| | $CP_3$ | – | – | – | | | |
| | $CP_4$ | IF DATA = 1 THEN<br>ACC ← DIL,<br>INPUT ← 0,<br>STATE ← F. | IF DATA = 1 THEN<br>OUTPUT← 0, | STATE ← F. | | | |

### 5.6.7 Input/Output Instructions

During RWD and WWD, the processor waits until the end of the execute cycle to check to see if the incoming data is ready ($DATA = 1$) or if it has been accepted ($DATA = 1$). The transition to fetch state occurs only if these conditions are satisfied. If not, the processor waits (loops) in the execute state until the conditions are met. Hence, the number of cycles needed for these instructions depends on the speeds of I/O devices.

Table 5.5 lists the control signals as implied by the microoperations in Tables 5.3 and 5.4. Logic diagrams that implement an HCU can be derived from the control-signal information in Tables 5.2 and 5.5.

Figure 5.11 shows the implementation of the four-phase clock to generate $CP_1$, $CP_2$, $CP_3$, and $CP_4$. A 4-bit shift register is used in the implementation. The master clock is an oscillator that starts emitting clock pulses as soon as the power to the machine is turned on. When the START button on the console (discussed in the next section) is pushed, the RUN flip-flop and the MSB of the shift register are set to 1. The master clock pulse is used to circulate the 1 in the MSB of the shift register through the other bits on the right, to generate the four clock pulses. The sequence of these pulses continues as long as RUN flip-flop is set. The HLT instruction resets the RUN flip-flop, thus stopping the four-phase clock. A "Master Clear" button on the console clears all the flip-flops when the RUN is not on.

A 2-to-4 decoder is used to generate F, D, and E signals corresponding to the fetch (00), defer (01), and execute (10) states. Figure 5.12 shows the state change circuitry and its derivation, assuming D flip-flops. The $CP_4$ is used as the clock for the state register, which along with the transition circuits is shown in Fig. 5.12 (page 250).

Figure 5.13 shows the circuit needed to implement the first three minor cycles of the fetch. The fourth minor cycle of fetch is implemented by the circuit in Fig. 5.14. Here, the RUN flip-flop is reset if the IR contains the HLT instruction. Indexing is performed only when $IR_{15}$ is a 0 and the instruction is not a TCA and not a HLT. Since the reset RUN flip-flop overrides any other operation, it is sufficient to implement the remaining condition ($IR_{15} = 0$ AND NOT TCA) for indexing. This condition is handled by the NOR gate. Corresponding to the four index register reference instructions, only the address portion of IR is transferred (without indexing) to MAR. The control signals corresponding to the other four instructions are similarly implemented.

Figure 5.15 shows the control circuits for the defer cycle and Fig. 5.16 for the execute cycle. The state transition signals of Fig. 5.12 are not repeated in these circuit diagrams, nor has the logic minimization been attempted in deriving these circuits.

**Table 5.5** (a) Control Signals for One-Address Instructions

| Machine cycle | Minor cycle | STA | ADD | BRU | LDX | STX |
|---|---|---|---|---|---|---|
| EXECUTE | $CP_1$ | – | READ. | MAR to BUS1, TRA1, BUS3 to PC. | READ. | – |
| | $CP_2$ | – | – | – | – | – |
| | $CP_3$ | ACC to BUS1, TRA1, BUS3 to MBR, WRITE. | ACC to BUS1, MBR to BUS2, ADD, BUS3 to ACC. | – | MBR to BUS2, TRA2, BUS3 to INDEX. | INDEX to BUS2, TRA2, BUS3 to MBR, WRITE. |
| | $CP_4$ | F to STATE. | F to STATE. | F to STATE. | F to STATE. | F to STATE. |

| Machine cycle | Clock pulse | TIX | TDX | BIP | BIN |
|---|---|---|---|---|---|
| EXECUTE | $CP_1$ | 1 to BUS1, INDEX to BUS 2, ADD, BUS3 to INDEX. | –1 to BUS1, INDEX to BUS2, ADD, BUS3 to INDEX. | – | – |
| | $CP_2$ | IF ZERO-INDEX = 0 THEN MAR to BUS1, TRA1, BUS3 to PC. | IF ZERO-INDEX $\neq$ 0 THEN MAR to BUS1, TRA1, BUS3 to PC. | – | – |
| | $CP_3$ | – | – | – | – |
| | $CP_4$ | F to STATE. | F to STATE. | IF ACCUMULATOR – POSITIVE = 1 THEN MAR to BUS1, TRA1, BUS3 to PC, F to STATE. | IF N = 1 THEN MAR to BUS1 TRA1, BUS3 to PC, F to STATE. |

**Table 5.5** (b) Control Signals for Zero-address Instructions

| Machine cycle | Clock pulse | RWD | WWD | TCA | SHR | SHL | HLT |
|---|---|---|---|---|---|---|---|
| FETCH | $CP_1$ | | | | | | |
| | $CP_2$ | Same as those for LDA in Table 5.2 | | | | | |
| | $CP_3$ | | | | | | |
| | $CP_4$ | 0 to DATA,<br>1 to INPUT,<br>E to STATE. | 0 to DATA,<br>1 to OUTPUT,<br>E to STATE.<br>ACC to DOL. | E to STATE. | ACC to BUS1,<br>SHR,<br>BUS3 to ACC. | ACC to BUS1,<br>SHL,<br>BUS3 to ACC. | 0 to RUN. |
| EXECUTE | $CP_1$ | – | – | ACC to BUS1,<br>COMP,<br>BUS3 to ACC. | | | |
| | $CP_2$ | – | – | ACC to BUS1,<br>1 to BUS2,<br>ADD,<br>BUS3 to ACC. | Execution cycle not needed; remains in F state. | | |
| | $CP_3$ | – | – | – | | | |
| | $CP_4$ | IF DATA = 1<br>THEN DIL<br>to ACC,<br>0 to INPUT,<br>F to STATE. | IF DATA = 1<br>THEN 0 to<br>OUTPUT,<br>F to STATE. | F to STATE. | | | |

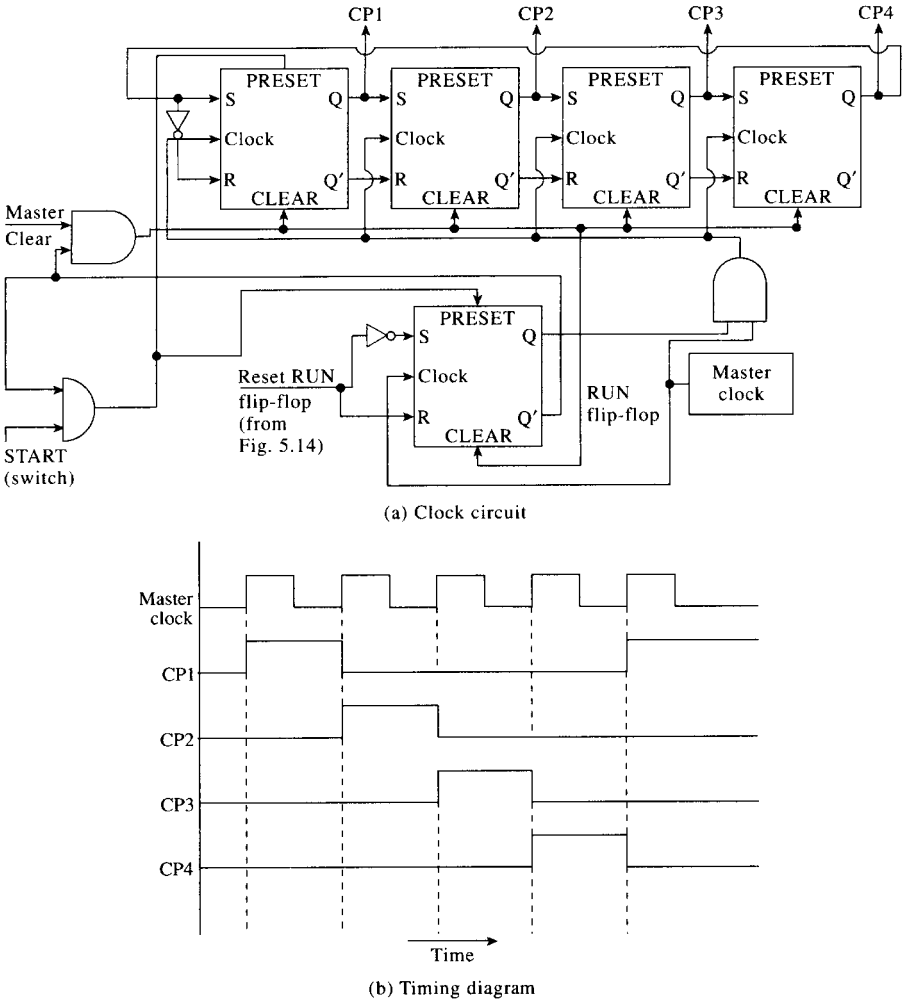
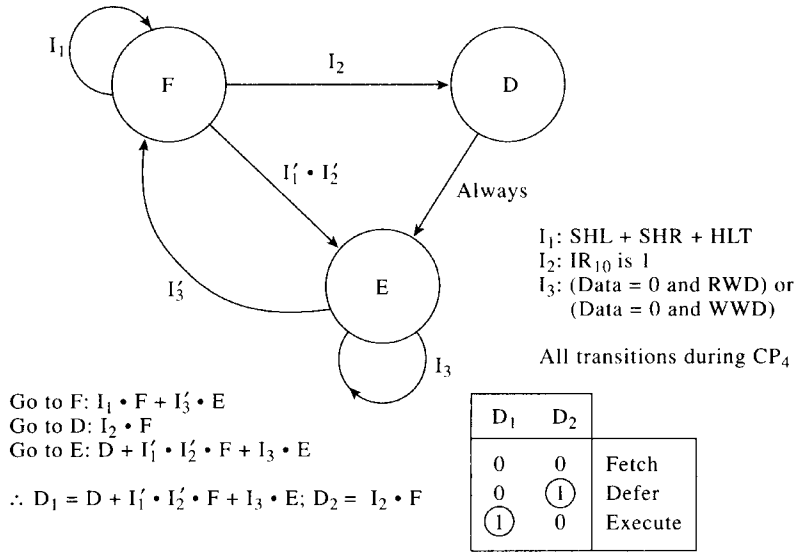



**Figure 5.11** Four-phase clock using SR flip-flops

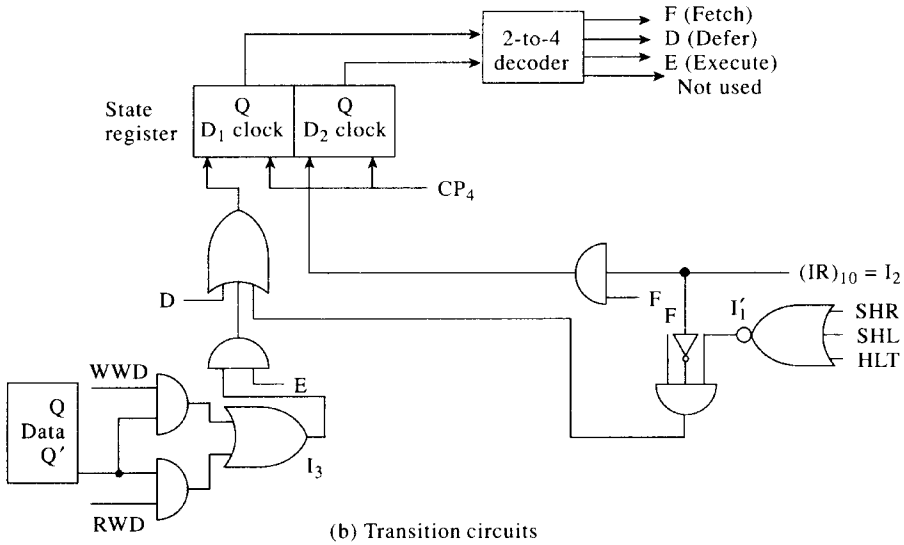
## 5.7 CONSOLE

We have included the design of the console here for completeness. This section can be skipped without a loss of continuity. We will not consider console operation, however, in the MCU design in the next section.

Figure 5.17 shows the ASC console. The console (control panel) enables the operator to control the machine. It can be used for loading programs and data into memory, observing the contents of registers and


I1
F
I2
D
I1' • I2'
Always
I3'
E
I3
I1: SHL + SHR + HLT
I2: IR10 is 1
I3: (Data = 0 and RWD) or
(Data = 0 and WWD)
All transitions during CP4
Go to F: I1 • F + I3' • E
Go to D: I2 • F
Go to E: D + I1' • I2' • F + I3 • E
∴ D1 = D + I1' • I2' • F + I3 • E; D2 = I2 • F
D1 D2
0 0 Fetch
0 1 Defer
1 0 Execute

(a) State register and transitions




(b) Transition circuits

**Figure 5.12** State register and transition circuits

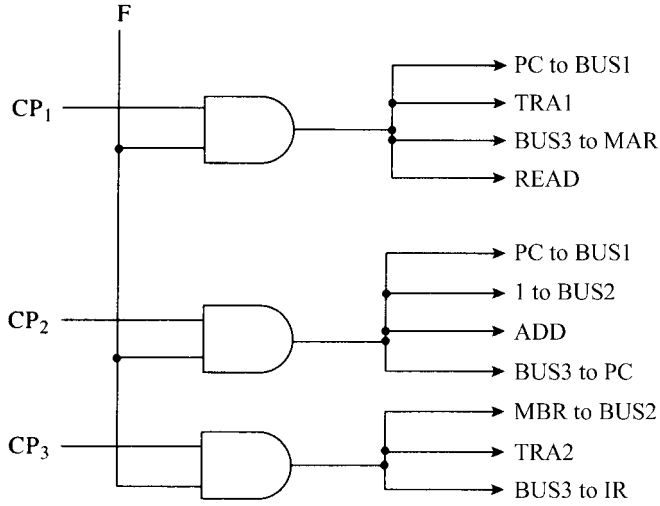


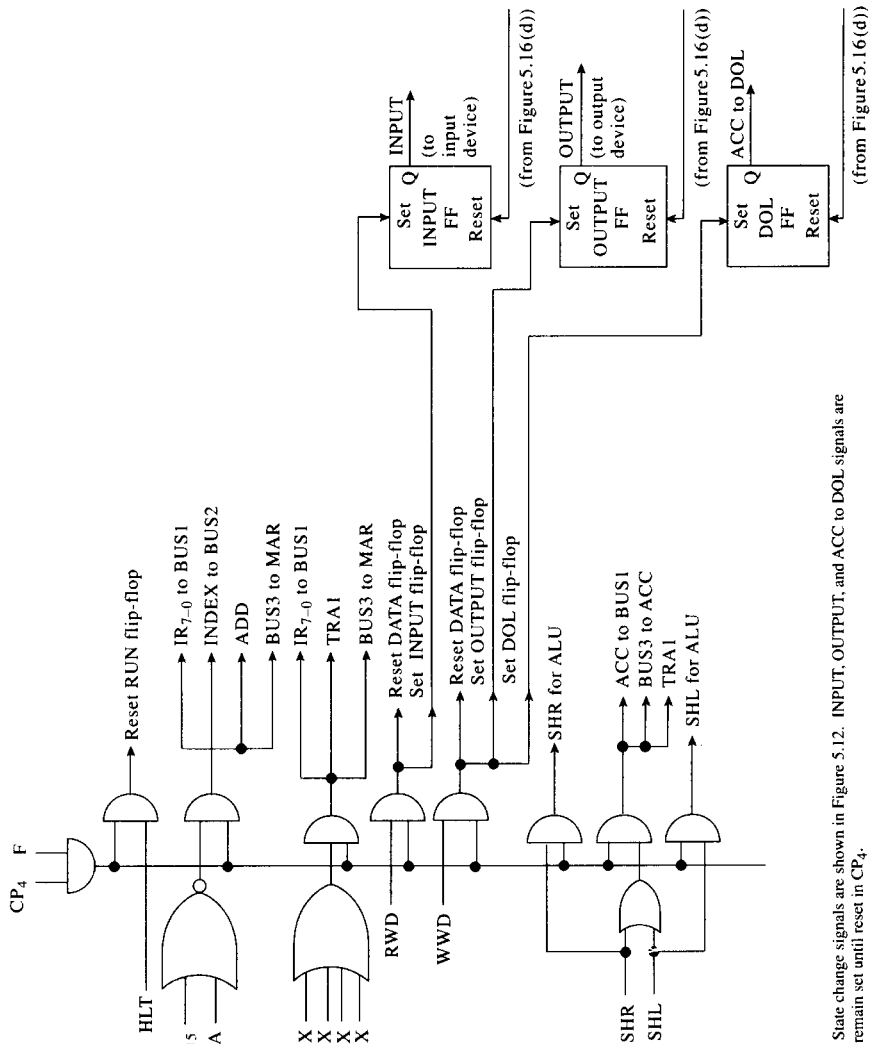

**Figure 5.13** Fetch cycle ($CP_1 - CP_3$)

memory locations, and starting and stopping the machine. There are sixteen lights (monitors) on the console that can display the contents of a selected register or memory location. There is a *switch bank* consisting of sixteen two-position switches that can be used for loading a 16-bit pattern into either PC or a selected memory location. There are two LOAD switches: LOAD PC and LOAD MEM. When LOAD PC is pushed, the bit pattern set on the switch bank is transferred to PC. When LOAD MEM is pushed, the contents of the switch bank are transferred to the memory location addressed by PC. There is a set of DISPLAY switches to enable the display of contents of ACC, PC, IR, index registers, PSR, or the memory location addressed by PC. The DISPLAY and LOAD switches are push-button switches and are mechanically ganged together so that only one of these switches is pushed (operative) at any time. The switch that was previously pushed pops out when a new switch is pushed. The MASTER CLEAR switch clears all ASC registers. The START switch sets the RUN flip-flop, which in turn starts the four-phase clock. There is also a POWER ON/OFF switch on the console.

Each switch on the console invokes a sequence of microoperations. Typical operations possible using the console along with the sequences of microoperations invoked are listed here:

Loading PC. Set the switch bank to the required 16-bit pattern.

LOAD PC PC ← Switch bank.




Note: State change signals are shown in Figure 5.12. INPUT, OUTPUT, and ACC to DOL signals are set and remain set until reset in $CP_4$.

**Figure 5.14** Fetch cycle ($CP_4$)

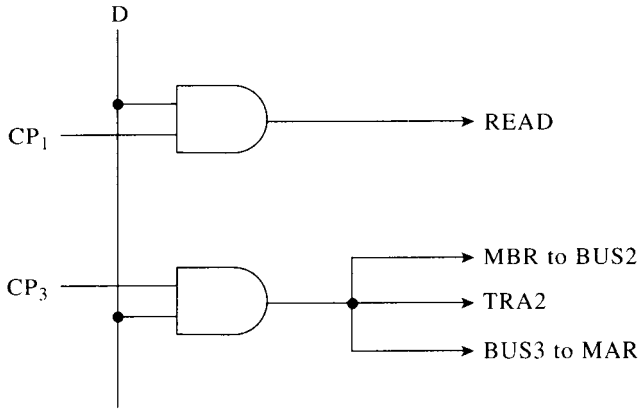



**Figure 5.15** Defer cycle

Loading memory. Load PC with the memory address.
Set switch bank to the data to be loaded into the memory location.

| | |
|---|---|
| LOAD MEM | MAR ← PC. |
| | MBR ← Switch Bank |
| | WRITE. |
| | PC ← PC + 1. |

### Display registers

Each of the register contents can be displayed on monitors (lights) by pushing the corresponding display switch.

MONITOR ← Selected register

Display memory. Set PC to the memory address.

| | |
|---|---|
| DISPLAY MEM | MAR ← PC, READ. |
| | PC ← PC + 1. |
| | MONITOR ← MBR. |

While loading and displaying memory, the PC value is incremented by 1 at the end of a load or a display. This enables easier loading and monitoring of the consecutive memory locations. To execute a program, the program and data are first loaded into the memory, PC is set to the address of the first executable instruction, and the execution is started by pushing the START switch. Since the content of any register or memory location can be placed on BUS3, a 16-bit *monitor* register is connected to BUS3. The lights

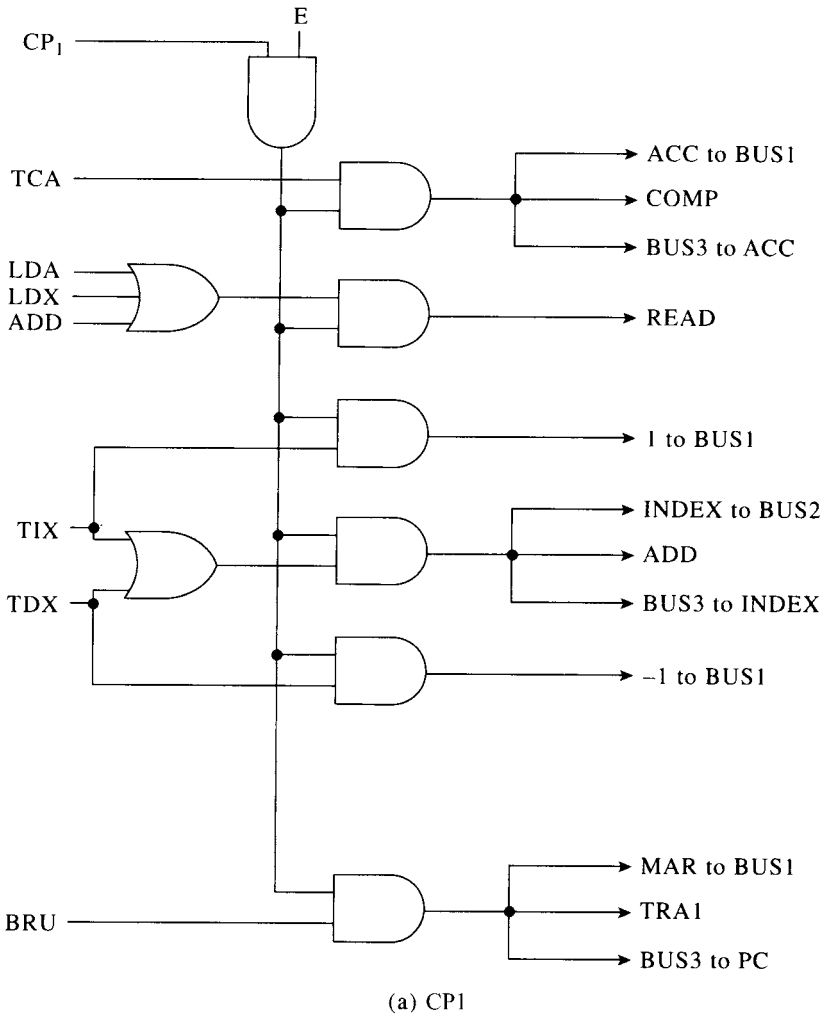
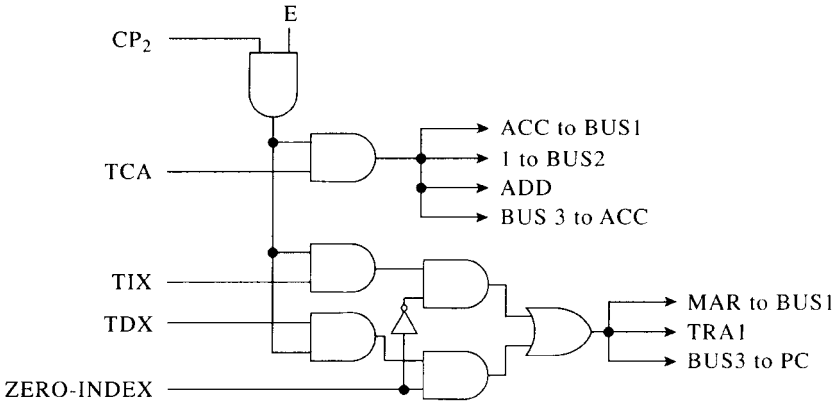


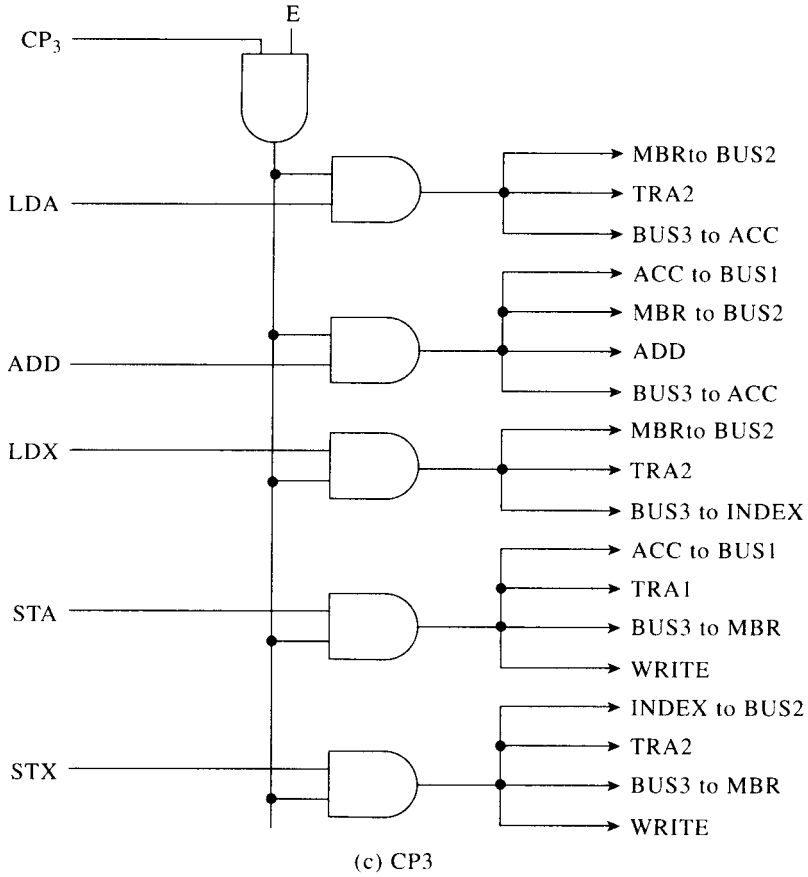

**Figure 5.16** Execute cycle

on the console display the contents of this register. The switch bank is connected to BUS2.

Figure 5.18 shows the control circuitry needed to display memory. The console is active only when the RUN flip-flop is RESET. When one of the LOAD or DISPLAY switches is depressed, the *console active* flip-flop is set for a time period equal to three clock pulses (all the console functions can be completed in three-clock-pulse time). The RUN flip-flop is also set during



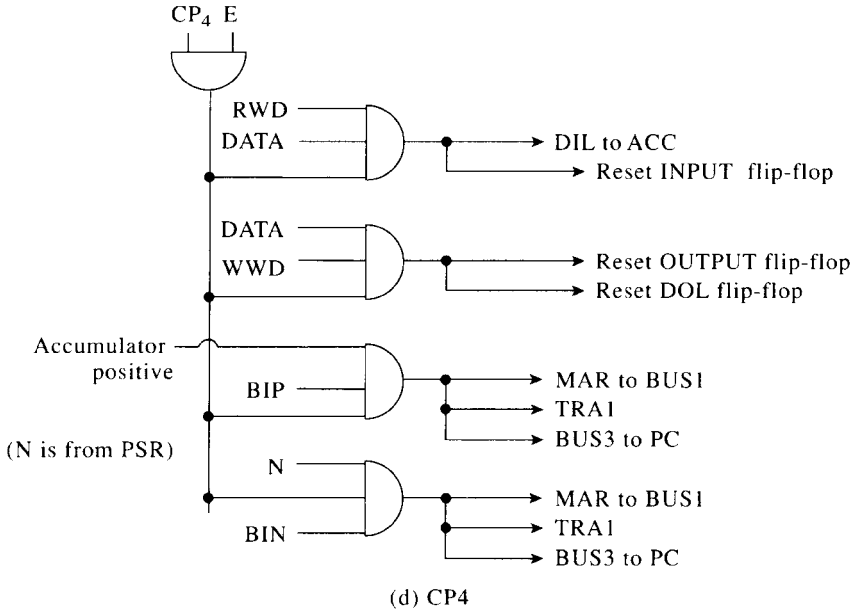

(b) CP2




(c) CP3

(*continued*)

**Figure 5.16** (Continued)




(d) CP4

**Figure 5.16** (Continued)

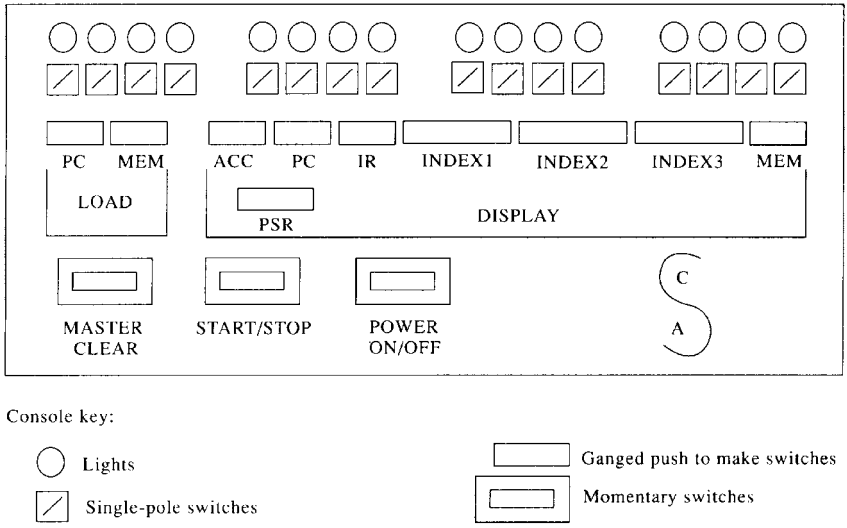



**Figure 5.17** ASC console

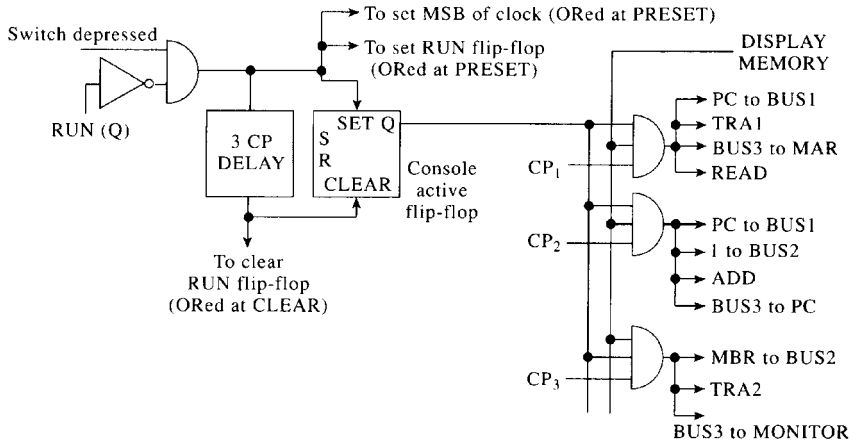



**Figure 5.18** Console circuits for DISPLAY MEMORY

this period, thus enabling the clock. The clock circuitry is active long enough to give three pulses and deactivated by resetting the RUN flip-flop. The START/STOP switch complements the RUN flip-flop each time it is depressed, thus starting or stopping the clock. Note that except for the START/STOP switch, the console is inactive when the machine is RUNning. Circuits to generate control signals corresponding to each LOAD and DISPLAY switch must be included to complete the console design. The console designed here is very simple compared to the consoles of machines available commercially.

## 5.8 MICROPROGRAMMED CONTROL UNIT

A hardwired control unit requires extensive redesign of the hardware if the instruction set has to be expanded or if the function of an instruction has to be changed. In practice, a flexible control unit is desired to enable tailoring the instruction set to the application environment. A microprogrammed control unit (MCU) offers such a flexibility. In an MCU, microprograms corresponding to each instruction in the instruction set are stored in a ROM called control ROM (CROM). A microcontrol unit ($\mu$CU) executes the appropriate microprogram based on the instruction in IR. The execution of a microinstruction is equivalent to the generation of control signals to bring out that microoperation. The $\mu$CU is usually hardwired and is comparatively simple to design since its function is only to execute microprograms in the CROM.

Figure 5.19 shows a block diagram of an MCU. Microprograms corresponding to fetch, defer, and execute cycles of each instruction are stored in the CROM. The beginning address of the fetch sequence is loaded into $\mu$MAR when the power is turned on. Then the CROM transfers the first microinstruction of fetch into $\mu$MBR. The $\mu$CU decodes this microinstruction to generate control signals required to bring about that microoperation. The $\mu$MAR is normally incremented by 1 at each clock pulse to execute the next microinstruction in sequence. This sequential execution is altered at the end of the fetch microprogram, since the execution of the microprogram corresponding to the execute cycle of the instruction now residing in IR must be started. Hence the $\mu$MAR must be set to the CROM address where the appropriate execute microprogram begins. At the end of execution of each execute microprogram, control is transferred to the fetch sequence. The function of the $\mu$CU is thus to set the $\mu$MAR to the proper value; i.e., the current value incremented by 1 or a jump address depending on the opcode and the status signals such as ZERO-INDEX, N, Z, etc.

For illustration purposes, a typical microinstruction might be PC $\leftarrow$ PC + 1. When this microinstruction is brought into $\mu$MBR, the decoder circuits generate the following control signals: PC to BUS1, 1 to BUS2, ADD, BUS3 to PC, and the $\mu$MAR is incremented by 1.

We will describe the design of an MCU for ASC next. Refer to Chapter 9 for further details on microprogramming.

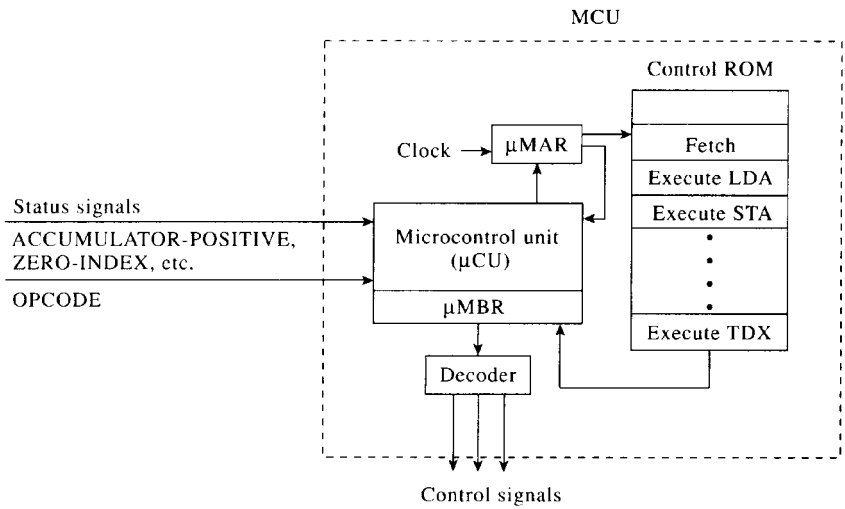


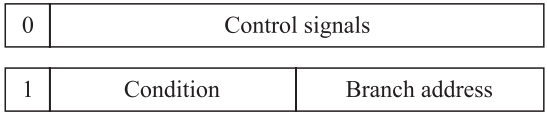

**Figure 5.19** Microprogrammed control unit (MCU) model

### 5.8.1 MCU for ASC

Table 5.6 shows the complete microprogram for ASC. The microprogram resembles any high-level language program and consists of "executable microinstructions" (which produce control signals as the result of execution) and "control microinstructions" (which change the sequence of execution of the microprogram based on some conditions). The first 32 microinstructions are merely jumps to the microinstruction sequences corresponding to the 32 possible instructions in ASC. Location 0 contains an infinite loop corresponding to the execution of HLT instruction. The START switch on the console takes ASC off this loop. That is, when the START switch is depressed, the microprogram begins execution at 32 where the fetch sequence begins. As soon as the instruction is fetched into IR, the microprogram jumps to one of the first 32 locations (based on the opcode in IR) and in turn to the appropriate execution sequence.

The indexing and indirect address computation are each now part of the execution sequence of instructions that use those addressing modes. As in HCU, the index register is selected by the IR decoder circuit and if the index flag corresponds to 00, no index register will be selected. The microinstruction sequence corresponding to indirect address computation (i.e., locations 38–40) is repeated for each instruction that needs it, for simplicity. (Alternatively, this sequence could have been written as a subprogram that could be called by each instruction sequence as needed.) At the end of each instruction execution, the microprogram returns to the fetch sequence.

In order to represent the microprogram in a ROM, the microprogram should be converted into binary. This process is very similar to an assembler producing an object code. There are two types of instructions in the microprogram, as mentioned earlier. To distinguish between the two types we will use a microinstruction format with a 1-bit microopcode. A microopcode of 0 indicates type 0 instructions that produce the control signals while microopcode 1 indicates type 1 instructions (jumps) that control the microprogram flow. The formats of the two types of microinstructions are shown below:




Each bit of the type 0 instruction can be used to represent a control signal. Thus, when this microinstruction is brought to $\mu$MBR, each nonzero bit in the microinstruction would produce the corresponding control signal. This

**Table 5.6** ASC Microprogram

| Address | | Microinstruction | Comments |
|---|---|---|---|
| 0 | | GO to 0 | Halt loop |
| 1 | | GO TO LDA | |
| 2 | | GO TO STA | |
| 3 | | GO TO ADD | |
| 4 | | GO TO TCA | |
| 5 | | GO TO BRU | Jump to |
| 6 | | GO TO BIP | appropriate |
| 7 | | GO TO BIN | execute |
| 8 | | GO TO RWD | routine |
| 9 | | GO TO WWD | |
| 10 | | GO TO SHL | |
| 11 | | GO TO SHR | |
| 12 | | GO TO LDX | |
| 13 | | GO TO STX | |
| 14 | | GO TO TIX | |
| 15 | | GO TO TDX | |
| 16 | | – | |
| 17 | | Reserved for other opcodes | |
| 18 | | – | |
| – | | – | |
| – | | – | |
| 31 | | – | |
| 32 | FETCH | MAR ← PC, READ. | Fetch begins |
| 33 | | PC ← PC + 1. | |
| 34 | | IR ← MBR. | Fetch ends |
| 35 | | GO TO* OPCODE. | Jump to 0 thru 31 based on OPCODE in $IR_{15-11}$ |
| 36 | LDA | MAR ← $IR_{7-0}$+ INDEX. | Index |
| 37 | | IF $IR_{10} = 0$ THEN GO TO M. | No indirect |
| 38 | | READ. | Indirect |
| 39 | | WAIT. | No operation, waiting for memory read |
| 40 | | MAR ← MBR. | Execute |
| 41 | M | READ. | |
| 42 | | WAIT. | |
| 43 | | ACC ← MBR. | |
| 44 | | GO TO FETCH. | End of LDA |
| 45 | STA | MAR ← $IR_{7-0}$+ INDEX. | Begin STA |
| 46 | L1 | IF $IR_{10} = 0$ THEN GO TO M1. | No indirect |
| 47 | | READ. | Indirect |

**Table 5.6** (Continued)

| Address | | Microinstruction | Comments |
|---|---|---|---|
| 48 | | WAIT. | |
| 49 | | MAR ← MBR. | |
| 50 | M1 | MBR ← ACC, WRITE. | |
| 51 | | WAIT. | |
| 52 | | GO TO FETCH. | End of STA |
| 53 | ADD | MAR ← $IR_{7-0}$ + INDEX | |
| 54 | L2 | IF $IR_{10} = 0$ THEN GO TO M2. | No indirect |
| 55 | | READ. | Indirect |
| 56 | | WAIT. | |
| 57 | | MAR ← MBR. | |
| 58 | M2 | READ. | |
| 59 | | WAIT. | |
| 60 | | ACC ← ACC + MBR. | |
| 61 | | GO TO FETCH. | End of ADD |
| 62 | TCA | ACC ← NOT (ACC). | |
| 63 | | ACC ← ACC + 1. | |
| 64 | | GO TO FETCH. | End of TCA |
| 65 | BRU | MAR ← $IR_{7-0}$+ INDEX. | |
| 66 | | IF $IR_{10} = 0$ THEN GO TO M3. | No indirect |
| 67 | | READ. | Indirect |
| 68 | | WAIT. | |
| 69 | | MAR ← MBR. | |
| 70 | M3 | PC ← MAR. | Execute BRU |
| 71 | | GO TO FETCH. | End BRU |
| 72 | BIP | IF ACC < = 0 THEN GO TO FETCH. | If ACC is not positive, go to fetch |
| 73 | | GO TO BRU. | |
| 74 | BIN | IF ACC > = 0 THEN GO TO FETCH. | If ACC is not negative, go to fetch |
| 75 | | GO TO BRU. | |
| 76 | RWD | DATA ← 0, INPUT ← 1. | Begin RWD |
| 77 | LL1 | WAIT. | Wait until data is ready |
| 78 | | IF DATA = 0 THEN GO TO LL1. | |
| 79 | | ACC ← DIL, INPUT ← 0. | Data to ACC |
| 80 | | GO TO FETCH. | End RWD |
| 81 | WWD | DATA ← 0, OUTPUT ← 1, DOL ← ACC. | Begin WWD |
| 82 | LL2 | WAIT. | Wait until data is accepted |
| 83 | | IF DATA = 0 THEN GO TO LL2. | |
| 84 | | OUTPUT ← 0. | |

(*continued*)

**Table 5.6** (Continued)

| Address | | Microinstruction | Comments |
|---|---|---|---|
| 85 | | GO TO FETCH. | End WWD |
| 86 | SHL | ACC ← SHL (ACC). | Shift left |
| 87 | | GO TO FETCH. | |
| 88 | SHR | ACC ← SHR (ACC). | Shift right |
| 89 | | GO TO FETCH. | |
| 90 | LDX | MAR ← $IR_{7-0}$. | Begin LDX |
| 91 | | IF $IR_{10} = 0$ THEN GO TO M6. | No indirect |
| 92 | | READ. | Indirect |
| 93 | | WAIT. | |
| 94 | | MAR ← MBR. | |
| 95 | M6 | READ. | Execute |
| 96 | | WAIT. | |
| 97 | | INDEX ← MBR. | |
| 98 | | GO TO FETCH. | End LDX |
| 99 | STX | MAR ← $IR_{7-0}$. | Begin STX |
| 100 | | IF $IR_{10} = 0$ THEN GO TO M7. | No indirect |
| 101 | | READ. | Indirect |
| 102 | | WAIT. | |
| 103 | | MAR ← MBR. | |
| 104 | M7 | MBR ← INDEX, WRITE. | Execute |
| 105 | | WAIT. | |
| 106 | | GO TO FETCH. | End STX |
| 107 | TIX | INDEX ← INDEX + 1. | Increment |
| 108 | | IF ZERO-INDEX = 1 THEN | ZERO-INDEX |
| 109 | LL3 | GO TO FETCH. | is non-zero |
| 110 | | MAR ← $IR_{7-0}$. | |
| 111 | | IF $IR_{10} = 0$ THEN GO TO M8. | No indirect |
| 112 | | READ. | Indirect |
| 113 | | WAIT. | |
| 114 | M8 | MAR ← MBR. | |
| 115 | | PC ← MAR. | Jump |
| 116 | TDX | GO TO FETCH. | |
| 117 | | INDEX ← INDEX – 1. | Decrement |
| | | IF ZERO-INDEX = 0 THEN | |
| 118 | | GO TO FETCH. | |
| | | GO TO LL3. | |

organization would not need any decoding of the microinstruction. The disadvantage is that the microinstruction word will be very long, thereby requiring a large CROM. One way to reduce the microinstruction word length is by encoding signals into several fields in the microinstruction, wherein a field represents control signals that are not required to be generated simultaneously. Each field is then decoded to generate the control signals.

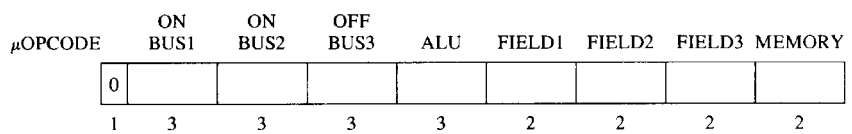
An obvious signal encoding scheme for ASC is shown in Fig. 5.20. Here, control signals are partitioned based on the busses they are associated with (i.e., on BUS1, ON BUS2, and OFF BUS3) and the ALU. The control signals to all these four facilities need to be generated simultaneously. This listing shows that each of the fields require three bits to represent the signals in that field. Figure 5.20(b) classifies the remaining

| Code | ON BUS1 | ON BUS2 | OFF BUS3 | ALU |
|---|---|---|---|---|
| 0 0 0 | None | None | None | None |
| 0 0 1 | ACC | INDEX | ACC | TRA1 |
| 0 1 0 | MAR | MBR | INDEX | TRA2 |
| 0 1 1 | $(IR)_{ADRS}$ | 1 | IR | ADD |
| 1 0 0 | PC | SWITCH BANK | MAR | COMP |
| 1 0 1 | 1 | | MBR | SHL |
| 1 1 0 | −1 | | PC | SHR |
| 1 1 1 | | | MONITOR | |

(a) 3-bit fields

| Code | FIELD1 | FIELD2 | FIELD3 | MEMORY |
|---|---|---|---|---|
| 0 0 | None | None | None | None |
| 0 1 | Not used | DIL to ACC | 0 to INPUT | READ |
| 1 0 | 1 to INPUT | ACC to DOL | 0 to OUTPUT | WRITE |
| 1 1 | 1 to OUTPUT | INHIBIT CLOCK | 0 to DATA | Not used |

(b) 2-bit fields

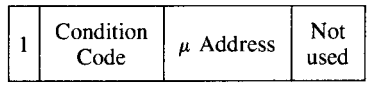



(c) Type 0 format

**Figure 5.20** Signal partitioning and encoding for microinstruction type 0

| Branching Condition | Condition Code | Comment |
|---|---|---|
| Unconditional branch | 000 | |
| GO TO* OPCODE | 001 | Branch based on opcode |
| Branch if $IR_{10} = 0$ | 010 | Indirect |
| Branch if ACC <= 0 | 011 | i.e., ACCUMULATOR-POSITIVE = 0 |
| Branch if ACC >= 0 | 100 | i.e., N = 0 |
| Branch if DATA = 0 | 101 | DATA flip-flop is reset. |
| Branch if ZERO-INDEX = 1 | 110 | Index Register content non-zero. |
| Branch if ZERO-INDEX = 0 | 111 | Index Register content is zero. |

(a) Type 1 microinstruction condition code field encoding




(b) Type 1 format

**Figure 5.21** Type 1 microinstruction encoding

control signals into four fields of two bits each. Thus the ASC type 0 instruction would be 21 bits long.

An analysis of the microprogram in Table 5.6 shows that there are 8 different branching conditions. These conditions are shown in Fig. 5.21. A 3-bit field is required to represent these conditions. The coding of this 3-bit field is also shown in Fig. 5.21. Since there are a total of 119 microinstructions, a 7-bit address is needed to completely address the CROM words. Thus type 1 microinstructions can be only 11 bits long. But to retain the uniformity, we will use a 21-bit format for type 1 instructions also. The formats are shown in Fig. 5.21(b).

The microprogram of Table 5.6 must be encoded using the instruction formats shown in Figs. 5.20 and 5.21. Figure 5.22 shows some examples of coding.

The hardware structure to execute the microprogram is shown in Fig. 5.23. The $\mu$MAR is first set to point to the beginning of the fetch sequence (i.e., address 32). At each clock pulse, the CROM transfers the microinstruction at the address pointed to by the $\mu$MAR into $\mu$MBR. If the first bit of $\mu$MBR is 0, the control decoders are activated to generate control signals and $\mu$MAR is incremented by 1. If the first bit of $\mu$MBR is a 1, the condition code decoders are activated. Based on the condition code, the $\mu$MAR is updated as follows:

| Condition | $\mu$MAR receives: |
|---|---|
| **START** | **32** |
| 000 | Address field of $\mu$MBR |
| 001 | Opcode from IR |
| 010 thru 111 | Address field of $\mu$MBR if the corresponding condition is satisfied; otherwise, increment $\mu$MAR by 1. |

Refer to Figure 5.21 (a) for details on condition codes.

The console operations are also facilitated by the microcode. The console hardware should gate a 32 into $\mu$MAR and start the clock every time the START switch is activated and the RUN flip-flop is reset. Similarly, corresponding to the LOAD PC switch, the following microprogram is required:

120 LOAD PC  PC ← SWITCH BANK.
121  GO TO 0.

Again, at each activation of LOAD PC switch, the address 120 should be gated into $\mu$MAR and clock started. At the end of LOAD PC the machine goes into halt state. The detailed design of console circuits is left as an exercise.

In an MCU, the time required to execute an instruction is a function of the number of miroinstructions in the corresponding sequence. The MCU returns to fetch the new instruction once the sequence is executed (the hardwired HCU waited until $CP_4$ for a state change). The overall operation is, however, slower than that of an HCU since each microinstruction is to be retrieved from the CROM and the time required between two clock pulses is thus equal to

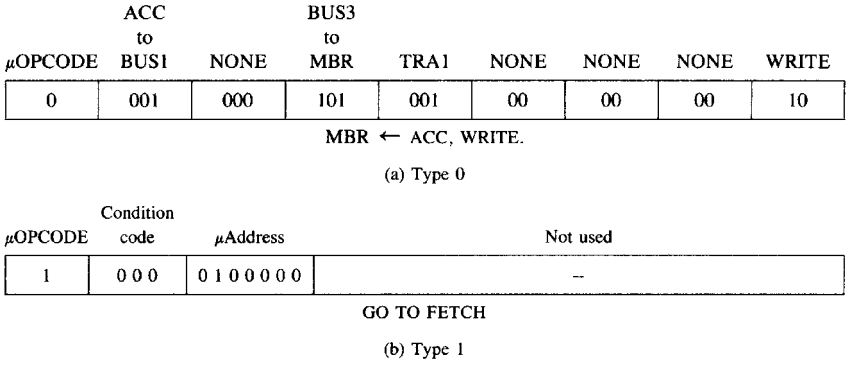



**Figure 5.22** Typical encoded microinstructions

```
LET
      E1 = (μopcode)'
      E2 = CC1
      E3 = START • RUN'
      E4 = CCØ +                          Unconditional
           CC2 • IR'10 +                  Indirect
           CC3 • (N + Z) +                ACC <= 0
           CC4 • (N' + Z) +               ACC >= 0
           CC5 • DATA' +                  DATA flip-flop reset
           CC6 • ZERO-INDEX +             INDEX ≠ 0
           CC7 • (ZERO-INDEX)'            INDEX = 0.

           where, CCØ–CC7 are decoded condition codes.
THEN
      E1:  μMAR ← μMAR + 1
      E2:  μMAR ← 00¢ OPCODE              (from IR)
      E3:  μMAR ← 32
      E4:  μMAR ← Address portion of μMBR.
```

**Figure 5.23** Microcontrol unit hardware

Slowest register transfer time + CROM access time.

The function of an instruction is easily altered by changing the microprogram. This requires a new CROM. Other than this, no hardware changes are needed. This attribute of an MCU is used in the "emulation" of computers where an available machine (host) is made to execute another (target) machine's instructions by changing the control store of the host.

In practice, CROM size is a design factor. The length of the microprogram must be minimized to reduce CROM size, thereby reducing the cost of MCU. This requires that each microinstruction contain as many microoperations as possible, thereby increasing the CROM word size, which in turn increases the cost of MCU. A compromise is thus needed. As in ASC-MCU, each microinstruction has to be encoded and the number of each encoded field in an instruction has to be kept low to reduce the CROM width. A microoperation then will span more than CROM word. Facilities are then needed to buffer the CROM words corresponding to a microoperation so that all required control signals are generated simultaneously. In a *horizontally microprogrammed* machine, each bit of the CROM word corresponds to a control signal. That is, there is no encoding of bits. Hence, the execution of the microprogram is fast since no decoding is necessary. But the CROM words are wide. In a *vertically microprogrammed* machine, the bits in the CROM word are encoded to represent control signals. This results in shorter

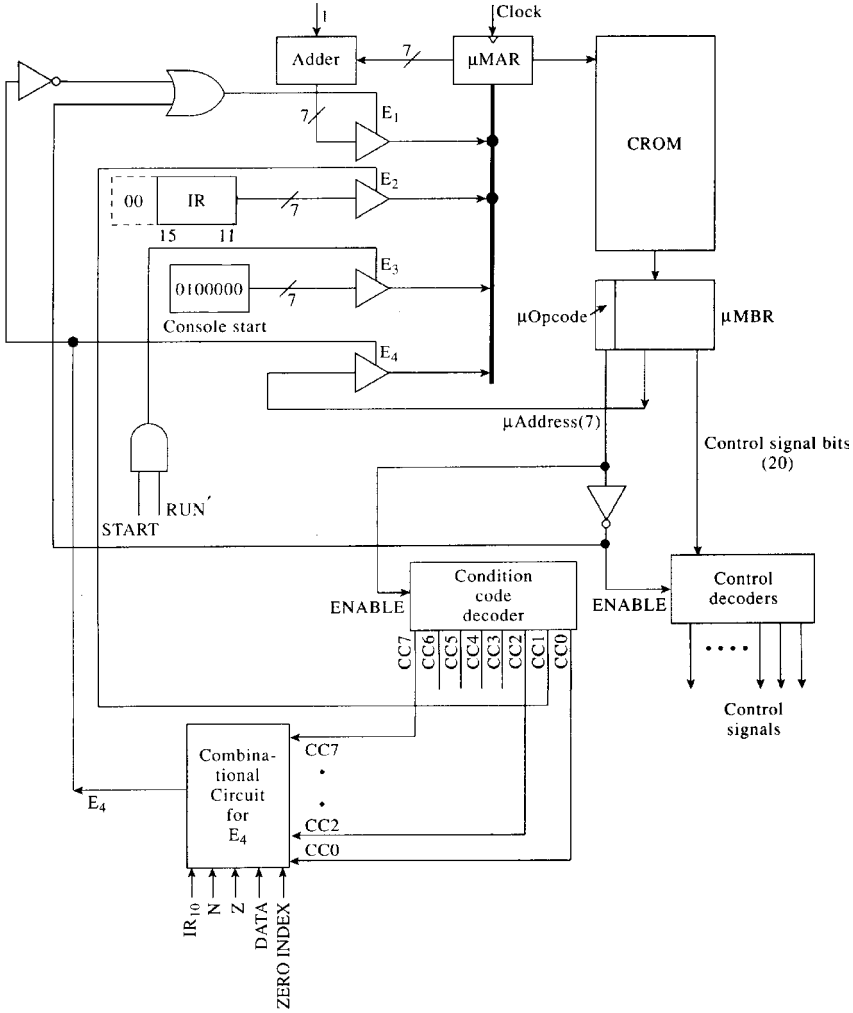



**Figure 5.23** (Continued)

words, but the execution of the microprograms becomes slow due to decoding requirements.

We conclude this section with the following procedure for designing an MCU, once the bus structure of the machine is established:

1. Arrange the microinstruction sequences into one complete program (microprogram).

2. Determine the microinstruction format(s) needed. The generalized format is as shown below:

| CONDITION | BRANCH ADDRESS | CONTROL SIGNALS |
|---|---|---|

This format might result in less efficient usage of the microword since BRANCH ADDRESS field is not used in those microinstructions that generate control signals and increment $\mu$MAR by 1 to point to the next microinstruction.

3. Count the total number of control signals to be generated. If this number is large, thus making the microinstruction wordlength too large, partition the control signals into distinct groups. Each partition contains control signals that need to be generated at the same time. Control signals that need to be generated simultaneously should be in separate groups. Assign fields in the control signal portion of the microinstruction format and encode them.
4. After the encoding in step 3, determine the $\mu$MAR size from the total number of microinstructions needed.
5. Load the microprogram into the CROM.
6. Design the circuit to initialize and uptake the contents of the $\mu$MAR.

## 5.9 SUMMARY

The detailed design of ASC provided here illustrates the sequence of steps in the design of a digital computer. In practice, the chronological order shown here can not be strictly followed. Several iterations between the steps are needed in the design cycle before a complete design is obtained. Several architectural and performance issues arise at each step in the design process. We have ignored those issues in this chapter but address them in subsequent chapters of this book.

This chapter introduced the design of microprogrammed control units. Chapter 9 provides further details on MCUs. Almost all modern-day computers use MCUs, because of the flexibility needed to update processors quickly, to meet the market needs.

## REFERENCES

Baron, R. J. and Higbie, L. *Computer Architecture*, Reading, MA: Addison Wesley, 1992.

Mano, M. M. *Computer Systems Architecture*, Englewood Cliffs, NJ: Prentice-Hall, 1992.

Patterson, D. A. and Hennessey, J. L. *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann, 1990.
Stone, H. S. *High Performance Computer Architecture*, Reading, MA: Addison Wesley, 1993.
Tannenbaum, A. S. and Goodman, J. R. *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall, 1998.

## PROBLEMS

5.1 Rewrite the control sequence for conditional branch instructions of ASC so that the instruction cycle terminates in one major cycle if the condition is not satisfied. Discuss the effect of this modification on the hardware of the control unit.

5.2 In a practical computer the shift instructions accommodate multiple shifts. Use the address field of SHR and SHL to specify the number of shifts. How can you accommodate this modification by

a. changes to hardwired control unit
b. changes to microprogram
c. changes to assembler.

5.3 Write register transfer sequences for the following new instructions on ASC. Subtract:

| | | |
|---|---|---|
| SUB | MEM | ACC ← ACC − M[MEM]. |

Load immediate:

| | | |
|---|---|---|
| LD1 | data | ACC ← data; assume data is in bits 0 thru 7 of the instruction. |

5.4 Write register transfer sequences for the following new instructions:

JSR: Subroutine jump; use memory location 0 for storing the return address.

RET: Return from subroutine.

5.5 Write the register transfer sequence needed to implement a multiply instruction on ASC. That is:

| | | |
|---|---|---|
| MPY | MEM | ACC ← ACC * M[MEM]. |

Use the repeated addition of ACC to itself M[MEM] times. Assume that the product fits in one word. List the hardware changes needed (if any).

5.6 Write microprograms to implement the solutions for Problems 5.3, 5.4 and 5.5

5.7 Write the microprogram to implement the following Move multiple words instruction:

MOV A,B

which moves $N$ words from memory locations starting at A to those at B. The value of $N$ is specified in the Accumulator. Assume that A and B are represented as 4-bit fields in the 8-bit operand field of the instruction, and indexing and indirecting are not allowed. What is the limitation of this instruction?

5.8 Implement the Move instruction of the problem above, by changes to the hardwired control unit.

5.9 Use two words to represent the Move instruction of Problem 5.7. The first word contains the opcode, address modifiers and the address of A and the second word contains the address modifiers and the address of B (the opcode field in this word is not used). Write the microprogram.

5.10 Discuss how you can accommodate more than one input/output device on ASC. Specify the hardware modifications needed.

5.11 Rewrite the microprogram sequence for HLT instruction, if the JSR and RET instructions of Problem 5.4 are implemented by extending the HLT opcode. That is, use the unused bits of the HLT instruction to signify JSR and RET.

5.12 Design a single-bus structure for ASC.

5.13 Code the microprogram segments for LDA and TIX instructions in Table 5.6 in binary and hexadecimal.

5.14 We have assumed single level of indirect addressing for ASC. What changes are needed to extend this to multiple levels whereby the indirection is performed until the most significant bit of the memory word, addressed at the current level, is non-zero.

5.15 Change the ASC instruction format to accommodate both preindexed and postindexed indirect addressing. Rewrite the microcode to accommodate those operations.

5.16 Use a 16-bit up-counter as the PC for ASC. What changes are needed for the control unit hardware? Does it alter the operation of any of the instructions?

5.17 Extend the JSR and RET instructions of Problems 5.4 to allow nested subroutine calls. That is, a subroutine can call another subroutine. Then the returns from the subroutines will be in a last-in-first-out order. Use a stack in your design. (Refer to Appendix D for details of stack implementation.)

5.18 Design a paged-addressing mechanism for ASC. Assume that the two most significant bits of the 8-bit direct address are used to select one of the four

10-bit segment registers. The contents of a segment register concatenated with the least significant six bits of the address field forms the 16-bit address. Show the hardware details. What is the effect of this addressing scheme on indirect and index address computations?

5.19 Do we still need the index and indirect addressing modes, after implementing the paged-addressing scheme of the problem above? Justify your answer.

5.20 Assume that the ASC memory is organized as 8 bits per word. That means each single-address instruction now occupies two words and zero-address instructions occupy one word each. The ASC bus structure remains the same. MBR is now 8 bits long and is connected to the least significant 8 bits of the bus structure. Rewrite the fetch microprogram.

5.21 Include four general-purpose registers into ASC structure to replace the accumulator and index registers. Each register can be used either as an accumulator or as an index register. Expand ASC instruction set to utilize these registers. Assume the same set of operations as are now available on ASC. Is there any merit in including register-to-register operations in the instruction set? Design the new instruction format.

5.22 If you are restricted to using only eight instructions to build a new machine, which of the sixteen instructions would you discard? Why?

5.23 How would you handle an invalid opcode in the microprogrammed control unit?

# 6
# Input/Output

In the design of ASC we assumed one input device and one output device transferring data in and out of the accumulator using a programmed input/output (I/O) mode. An actual computer system, however, consists of several input and output devices, or *peripherals*. Although the programmed I/O mode can be used in such an environment, it is slow and may not be suitable, especially when the machine is used as a real-time processor responding to irregular changes in the external environment. Consider the example of a processor used to monitor the condition of a patient in a hospital. Although the majority of its patient data gathering operations can be performed in a programmed I/O mode, alarming conditions such as abnormal blood pressure or temperature occur irregularly and detection of such events requires that the processor be interrupted by the event from its regular activity. We will discuss the general concept of interrupt processing and interrupt-driven I/O in this chapter.

The transfer of information between the processor and a peripheral consists of the following steps:

1. Selection of the device and checking the device for readiness
2. Transfer initiation, when the device is ready
3. Information transfer
4. Conclusion.

These steps can be controlled by the processor or the peripheral or both. Contingent upon where the transfer control is located, three modes of input/output are possible. They are

1. Programmed I/O
2. Interrupt mode I/O
3. Direct memory access (DMA).

We will discuss each of these modes in detail following a discussion of the general I/O model. Pertinent details of I/O structures of some popular computer systems are provided as examples.

## 6.1 GENERAL I/O MODEL

The *I/O structure* of ASC with one input device and one output device is shown in Fig. 6.1. ASC communicates with its peripherals through *data input lines* (DIL) and *data output lines* (DOL). There are two *control lines*: input and output. The *data flip-flop* in the control unit is used to coordinate the I/O activities.

Figure 6.2 shows the generalization of ASC I/O structure to include multiple I/O devices. To address multiple devices, a device number (or address) is needed. This device address can be represented in the 8-bit operand field of RWD and WWD instructions. In fact, since the Index and Indirect fields of these instructions are not used, device addresses as large as 11 bits can be used. In Fig. 6.2 it is assumed that only four bits are used to represent the device address. Thus, it is possible to connect 16 input and 16 output devices to ASC. A 4-to-16 decoder attached to device address bits is

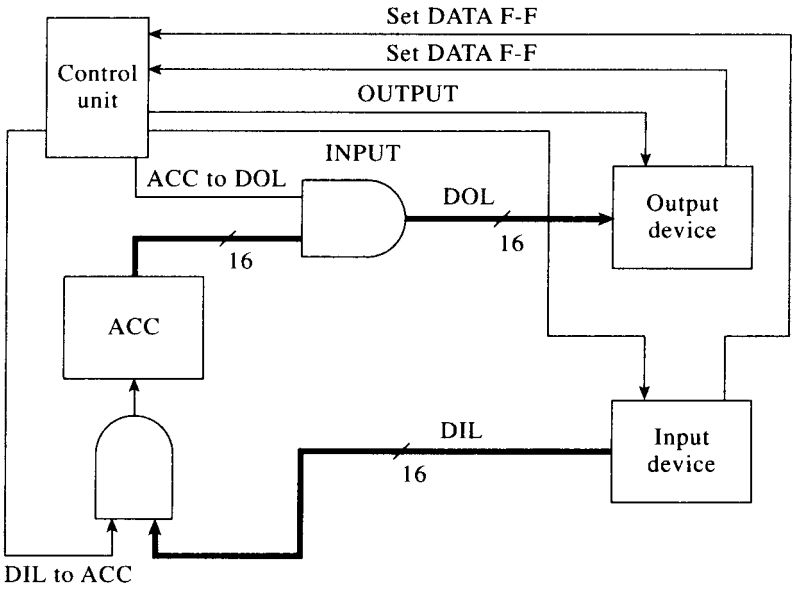



**Figure 6.1** ASC I/O structure

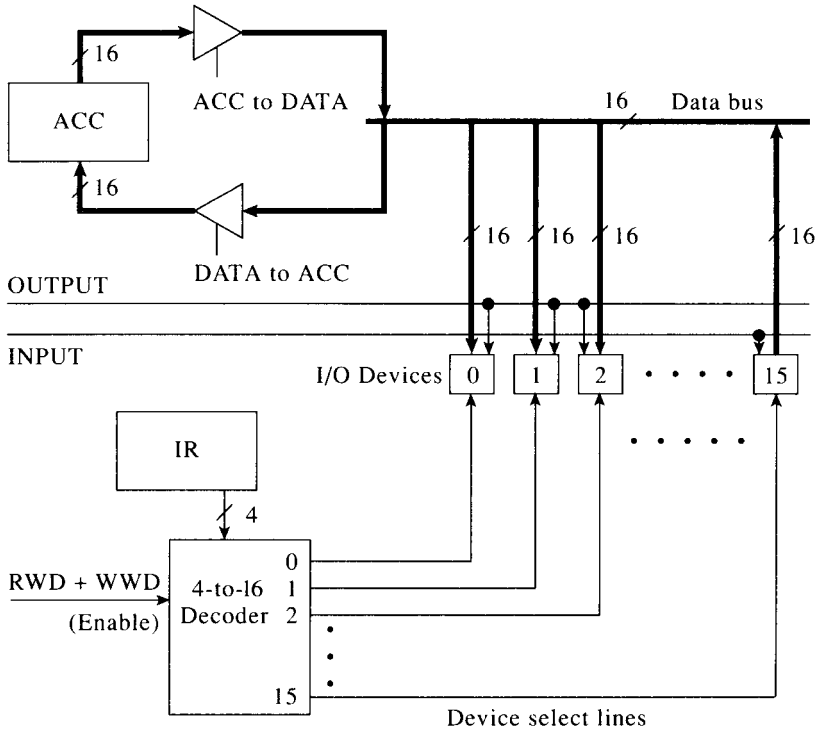



**Figure 6.2** Interfacing multiple devices to ASC

activated during RWD and WWD instruction cycles, to select one of the 16 input or output devices respectively. If the INPUT control line is active, the selected input device sends the data to the processor. If the OUTPUT control line is active, the selected output device receives the data from the processor. The DIL and DOL of Chapter 5 are replaced by the bidirectional data bus.

Figure 6.3 shows the generalized I/O structure used in practice. Here, the device address is carried on the *address bus* and is decoded by each device. Only the device whose address matches that on the address bus will participate in either input or output operation. The *data* bus is bidirectional. The *control bus* carries *control* signals (such as INPUT, OUTPUT, etc.). In addition, several *status signals* (such as DEVICE BUSY, ERROR) originating in the device interface also form part of the control bus.

In the structure shown in Fig. 6.3, the memory is interfaced to the CPU through a memory bus (consisting of address, data, and control/

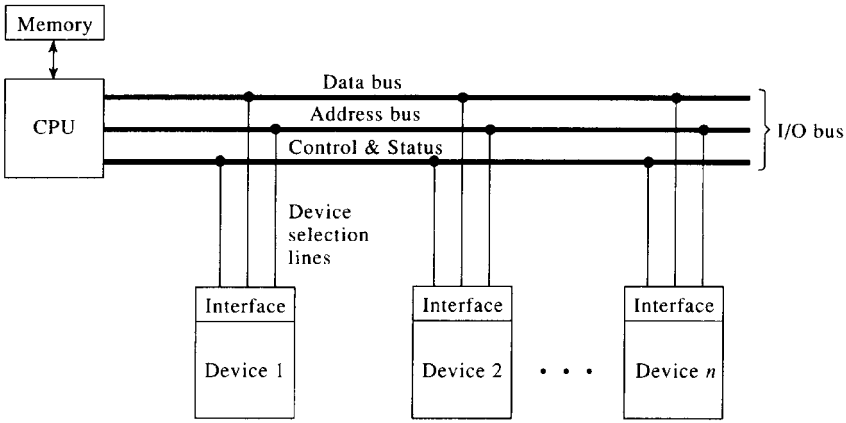



**Figure 6.3** General I/O structure

status lines), and the peripherals communicate with the CPU over the I/O bus. Thus there is a *memory address space* and a separate *I/O address space*. The system is said to use the *isolated I/O* mode. This mode of I/O requires a set of instructions dedicated for I/O operations. In some systems both the memory and I/O devices are connected to the CPU through the same bus, as shown in Fig. 6.4. In this structure, the device addresses are a part of the memory address space. Hence, the load and store instructions used with memory operands can be used as the read and write instructions with respect to those addresses configured for I/O devices. This mode of I/O is known as *memory-mapped I/O*. The advantage of memory-mapped I/O is that separate I/O instructions are not needed; the disadvantages are that some of the memory address space is used up by the I/O, and it can be difficult to distinguish between the memory and I/O-oriented operations in a program.

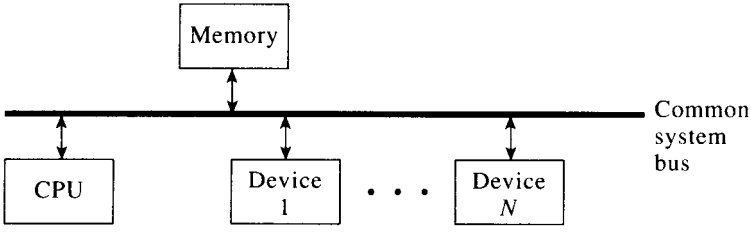



**Figure 6.4** Common bus (for both memory and I/O)

Although the above description implies two busses in the system structure for the isolated I/O mode, it need not be so in practice. It is possible to *multiplex* memory and I/O addresses on the same bus, while using control lines to distinguish between the two operations.

Figure 6.5 shows the functional details of a *device interface*. A device interface is unique to a particular device since each device is unique with respect to its data representation and read-write operational characteristics. The device interface consists of a controller that receives command (that is, control signals) from the CPU and reports the status of the device to the CPU. If the device is, for example, a tape reader, typical control signals are: IS DEVICE BUSY?, ADVANCE TAPE, REWIND, and the like. Typical status signals are: DEVICE BUSY, DATA READY, DEVICE NOT OPERATIONAL, and so on. Device selection (address decoding) is also shown as a part of the controller. The *transducer* converts the data represented on the I/O medium (tape, disk, etc.) into the binary format and stores it in the data *buffer*, if the device is an input device. In the case of an output device, the CPU sends data into the buffer and the transducer converts this binary data into a format suitable for output onto the external medium (for example, 0/1 bit write format onto a magnetic tape or disk, ASCII patterns for a printer, and so on).

Figure 6.6 shows the functional details of an interface between the CPU and a magnetic tape unit. The interface is very much simplified and illustrates only the major functions. When the device address on the ADDRESS bus corresponds to that of the tape device, the SELECT DEVICE signal becomes active. The CPU outputs the ADDRESS VALID control signal a little after the decoder outputs are settled, which is used to clock the DEVICE READY flip-flop. The DEVICE READY flip-flop is cleared since the output of the decoder is zero. The $Q'$ output of the

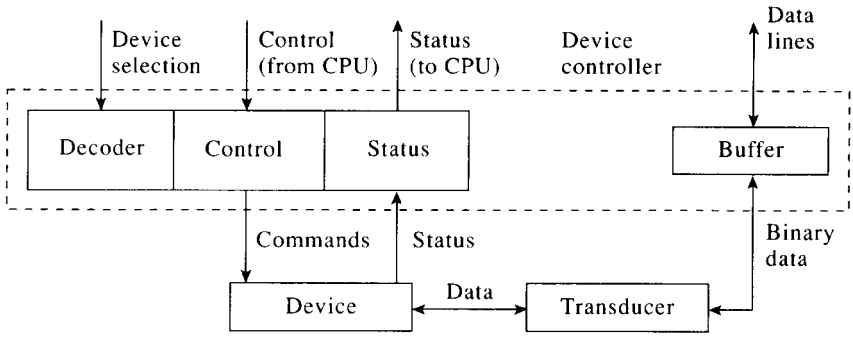



**Figure 6.5** Device interface

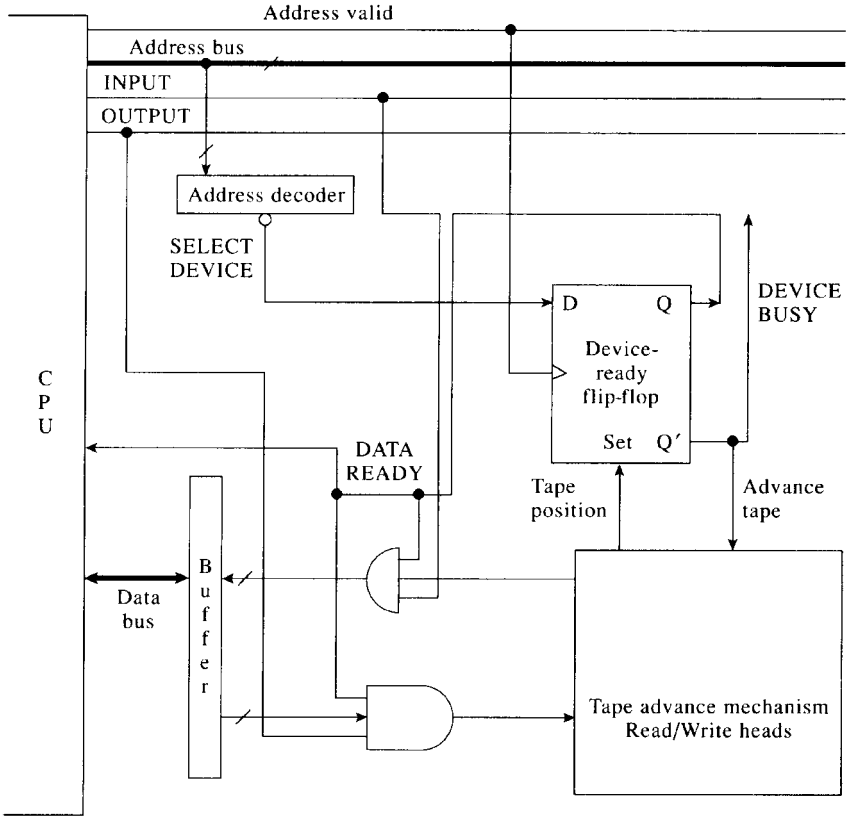



**Figure 6.6** Tape interface

DEVICE READY flip-flop then becomes the DEVICE BUSY status signal while the tape is being advanced to the next character position. Once the position is attained, the tape mechanism generates the TAPE POSITION signal, which sets the DEVICE READY flip-flop (through its asynchronous set input) and in turn generates the DATA READY signal for the CPU. During the data read operation, the INPUT signal being active, the device reads the data from the tape, loads it into its buffer, and gates it onto the data bus. In response to the DATA READY, the CPU gates the data bus into its accumulator.

During the data write, the CPU sends the data over the data bus into the device buffer, sets the OUTPUT control signal, sets the address bus with the device address and outputs the ADDRESS VALID signal. The opera-

tion of the DEVICE READY flip-flop is similar to that during the data read operation. The data is written onto the tape when the DATA READY signal becomes active. The DATA READY also serves as the data accepted signal for the CPU, in response to which the CPU removes the data and address from respective busses. We will describe the CPU-I/O-device handshake further in the next section.

## 6.2 THE I/O FUNCTION

The major functions of a device interface are

1. Timing
2. Control
3. Data conversion
4. Error detection and correction.

The timing and control aspects correspond to the manipulation of control and status signals to bring about the data transfer. In addition, the operating speed difference between the CPU and the device must be compensated for by the interface. In general, data conversion from one code to the other is needed, since each device (or the medium on which data are represented) may use a different code to represent data. Errors occur during transmission and must be detected and if possible corrected by the interface. A discussion of each of these I/O functions follows.

### 6.2.1 Timing

So far in this chapter we have assumed that the CPU controls the bus (that is, it is *bus master*). In general, when several devices are connected to the bus, it is possible that a device other than the CPU can become bus master. Thus, among the two devices involved in the data transfer on the bus, one will be the bus *master* and the other will be the *slave*. The sequence of operations needed for a device to become the bus master are described later in this chapter.

The data transfer on a bus between two devices can be either *synchronous* or *asynchronous*. Figure 6.7 shows the timing diagram for a typical synchronous bus transfer. The clock serves as the timing reference for all the signals. Either the rising or falling edge can be used. During the read operation shown in (a), the bus master activates the READ signal and places the ADDRESS of the slave device (the device from which it is trying to read data) on the bus. All the devices on the bus decode the address, and the device whose address matches the address on the bus responds by placing

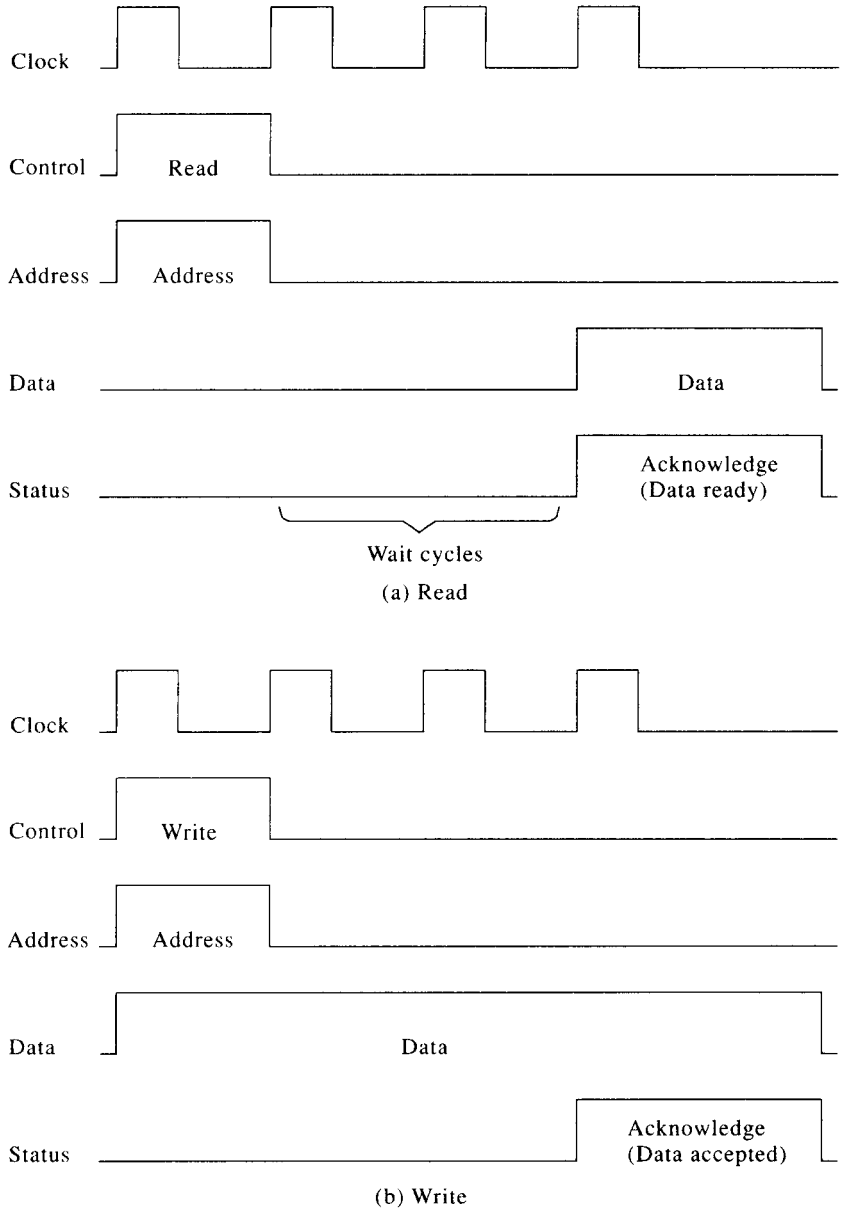



**Figure 6.7** Synchronous transfer

the data on the bus several clock cycles later. The slave may also provide *STATUS information* (such as error, no error, and so on) to the master. The number of clock cycles (wait cycles) required for this response depends on the relative speeds of master and slave devices. If the waiting period is known, the master can gate the data from the data bus at the appropriate time. In general, to provide a flexible device interface, the slave device is required to provide a control signal – ACKNOWLEDGE (abbreviated ACK) or DATA READY – to indicate the validity of data on the bus. Upon sensing the ACK, the master gates the data into its internal registers.

Figure 6.7(b) shows the timing for a synchronous WRITE operation. Here, the bus master activates the WRITE control signal and places the address of the slave device on the address lines of the bus. While the devices are decoding the address, the master also places the data on the DATA bus. After a wait period, the slave gates the data into its buffer and places the ACK (DATA ACCEPTED) on the bus, in respones to which the master removes the data and WRITE control signal from the bus.

Note that all the operations described above are synchronized with the clock. In an *asynchronous* transfer mode, the sequence of operations is the same as above except that there will be no clock.

Figure 6.8 shows the timing diagrams for an asynchronous transfer between the source and destination devices. In (a), the source initiates the transfer by placing data on the bus and setting DATA READY. The destination ACKs, in response to which the DATA READY signal is removed, after which the ACK is removed. Note that the data are removed from the bus only after the ACK is received. In (b) the destination device initiates (that is, requests) the transfer, in response to which the source puts the data on the bus. The acknowledge sequence is the same as in (a).

Peripheral devices usually operate in an asynchronous mode with respect to the CPU because they are not usually controlled by the same clock that controls the CPU. The sequence of events required to bring about the data transfer between two devices is called the *protocol* or the *handshake*.

### 6.2.2 Control

During the data transfer handshake, some events are brought about by the bus master and some are brought about by the slave device. The data transfer is completely controlled by the CPU in the programmed I/O mode. A typical protocol for this mode of I/O follows. We have combined the protocols for both input and output operations. A device will be either in the input or the output mode at any given time.

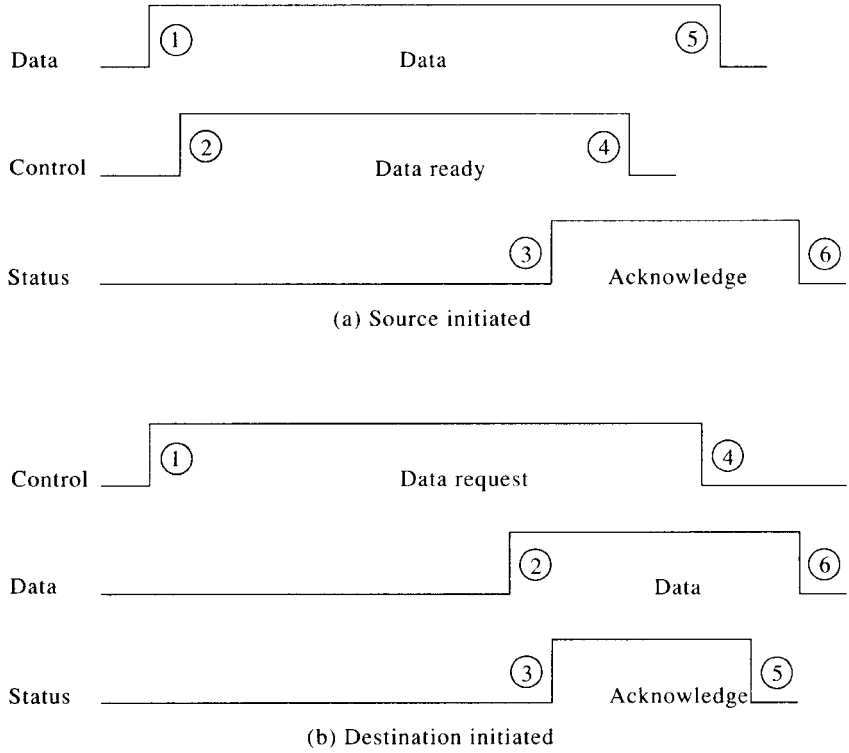



**Figure 6.8** Asynchronous transfer

| Processor | Device controller |
|---|---|
| 1. Selects the device and checks the status of the device. | |
| | 2. Signals the processor that it is ready or not ready. |
| 3. (If device not ready, go to step 1; if ready, go to step 4.) | |
| 4. Signals the device to initiate data transfer (send data or accept data); if OUTPUT, gates data onto data lines and sets output control line. | |
| | 5. If OUTPUT, signals the processor that the data are accepted; if INPUT, gathers data and signals CPU that the data are ready on data lines. |

| | |
|---|---|
| 6. If INPUT, accepts the data; if OUTPUT, removes data from data lines. | |
| 7. Disconnects the device (i.e., removes the device address from address lines). | |

This sequence repeats for each transfer. The speed difference between the CPU and the device renders this mode of I/O inefficient.

An alternative is to distribute part of the control activities to the device controller. Now, the CPU sends a command to the device controller to input or output data and continues its processing activity. The controller collects the data from (or sends data to) the device and "interrupts" the CPU. The CPU disconnects the device after the data transfer is complete (that is, the CPU services the interrupt) and returns to the mainline processing from which it was interrupted. A typical sequence of events during an *interrupt-mode I/O* is:

| Processor | Device controller |
|---|---|
| 1. Selects the device: if OUTPUT, puts the data on data lines and sets output control line; if INPUT, sets input control line. | |
| 2. Continues the processing activity. | |
| | 3. If OUTPUT, collects data from data lines and transmits to the medium; if INPUT, collects data from the medium into the data buffer. |
| | 4. Interrupts the processor. |
| 5. Processor recognizes the interrupt and saves its processing status; if OUTPUT, removes data from data lines; if INPUT, gates data into accumulator. | |
| 6. Disconnects the device. | |
| 7. Restores the processing status and returns to processing activity. | |

The protocol assumes that the CPU always initiates the data transfer. In practice, a peripheral device may first interrupt the CPU and the type of transfer (input or output) is determined during the CPU-peripheral handshake. The data input need not be initiated only by the CPU. The causes of interrupt and the popular interrupt structures are discussed in the next section. Interrupt mode I/O reduces the CPU wait time (for the slow device) but requires a more complex device controller than in the programmed I/O mode.

In addition to the above protocols, other control and timing issues are introduced by the characteristics of the *link* (that is, the data line or lines that connect the device and the CPU). The data link can be *simplex* (unidirectional), *half-duplex* (either direction, one way at a time), *full-duplex* (both directions simultaneously), *serial*, or *parallel*. Serial transmission requires that a constant clock rate be maintained throughout data transmission to avoid synchronization problems; in parallel transmission, care must be taken to avoid data "skewing" (i.e. data arriving at different times on the bus lines due to different electrical characteristics of individual bit lines).

Data transfer between peripheral devices located in the vicinity of the CPU is usually performed in parallel mode, while the remote devices communicate with the CPU in serial mode. We will assume parallel mode transfer in the following sections. Serial transfer mode is described further in Section 6.7.

### 6.2.3 Data Conversion

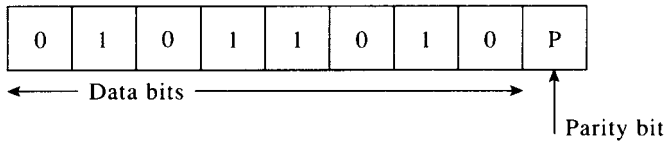
The data representation on the I/O medium is unique to each medium. For example, a magnetic tape uses either ASCII or EBCDIC codes to represent data. Internally, the CPU might use a binary or BCD (decimal) representation. In addition, the interface link might be organized as serial-by-bit, serial-by-character (quasi-parallel), or serial-by-word (fully parallel). Thus, two levels of data conversion are to be accomplished by the interface: conversion from peripheral to link format and from link to CPU format.

### 6.2.4 Error Detection and Correction

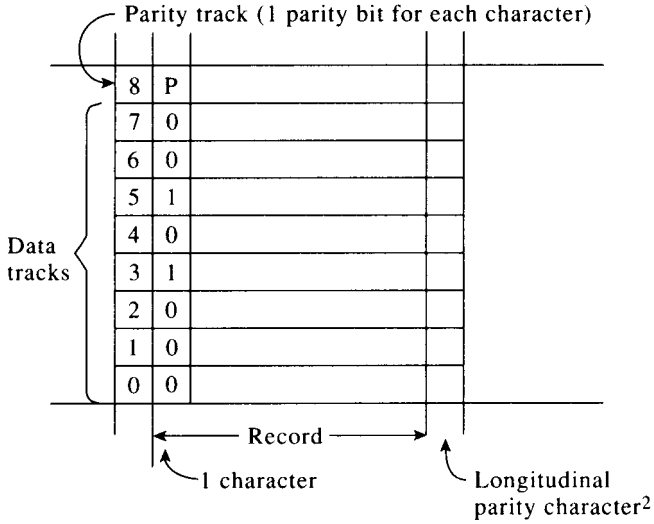
Errors may occur whenever data are transmitted between two devices. One or more extra bits known as *parity bits* are used as part of the data representation to facilitate error detection and correction. Such parity bits, if not already present in the data, are included into the data stream by the interface before transmission and checked at the destination. Depending on the number of parity bits, various levels of error detection and correction are

possible. Error detection and correction are particularly important in I/O because the peripheral devices exposed to the external environment are error prone. Errors may be due to mechanical wear, temperature and humidity variations, mismounted storage media, circuit drift, incorrect data transfer sequences (protocols), and the like.

Figure 6.9(a) shows a parity bit included into a data stream of 8 bits. The parity bit P is 1 if *odd parity* is used (the total number of 1s is odd) and 0




(a) Parity checking[1]




(b) Magnetic tape data representation

[1] P = 1 for odd parity (total number of 1s is odd); P = 0 for even parity (total number of 1s is even).
[2] One bit for each track; parity check for the complete record.

**Figure 6.9** Parity bits for error detection on a magnetic tape

if *even parity* is used (the total number of 1s is even). When this 9-bit data word is transmitted, the receiver of the data computes the parity bit. If P matches the computed parity, there is no error in transmission; if an error is detected, the data can be retransmitted.

Figure 6.9(b) shows a parity scheme for a magnetic tape. An extra track (track 8) is added to the tape format. The parity track stores the parity bit for each character represented on the eight tracks of the tape. At the end of a record, a longitudinal parity character consisting of a parity bit for each track is added. More elaborate error detection and correction schemes are often used; references listed at the end of this chapter provide details of those schemes.

## 6.3 INTERRUPTS

In a number of conditions the processor may be interrupted from its normal processing activity. Some of these conditions are:

1. Power failure as detected by a sensor
2. Arithmetic conditions such as overflow and underflow
3. Illegal data or illegal instruction code
4. Errors in data transmission and storage
5. Software-generated interrupts (as intended by the user)
6. Normal completion of asynchronous transfer.

In each of these conditions the processor must discontinue its processing activity, attend to the interrupting condition, and (if possible) resume the processing activity from where it had been when the interrupt occurred. In order for the processor to be able to resume normal processing after servicing the interrupt, it is essential to at least save the address of the instruction to be executed just before entering the interrupt service mode. In addition, contents of the accumulator and all other registers must be saved. Typically, when an interrupt is received, the processor completes the current instruction and jumps to an *interrupt service routine*. An interrupt service routine is a program preloaded into the machine memory that performs the following functions:

1. Disables further interrupts temporarily.
2. Saves the processor status (all registers).
3. Enables further interrupts.
4. Determines the cause of interrupt.
5. Services the interrupt.
6. Disables interrupts.

7. Restores processor status.
8. Enables further interrupts.
9. Returns from interrupt.

The processor disables further interrupts just long enough to save the status, since a proper return from interrupt service routine is not possible if the status is not completely saved. The processor status usually comprises the contents of all the registers, including the program counter and the program status word. "Servicing" the interrupt simply means taking care of the interrupt condition: in the case of an I/O interrupt, it corresponds to data transfer; in case of power failure, it is the saving of registers and status for a normal resumption of processing when the power is back; during an arithmetic condition, it is checking the previous operation or simply setting a flag to indicate the arithmetic error.

Once the interrupt service is complete, the processor status is restored. That is, all the registers are loaded with the values saved during step 2. Interrupts are disabled during this restore period. This completes the interrupt service, and the processor returns to the normal processing mode.

### 6.3.1 Interrupt Mechanism for ASC

An interrupt may occur at any time. The processor recognizes interrupts only at the end of the execution of the current instruction. If the interrupt needs to be recognized earlier (say, at the end of a fetch, before execution), a more complex design is needed because the processor status has to be rolled back to the end of the execution of the previous instruction. Assuming the simpler case for ASC, the fetch microsequence must be altered to recognize the interrupt. Let us assume that there is an interrupt input (INT) into the control unit.

In the following interrupt scheme for ASC, we assume that there will be only one interrupt at a time, that is, no interrupts will occur until the current interrupt is completely serviced. (We will remove this restriction later.) We will reserve memory locations 0 through 5 for saving registers (PC, ACC, index registers, and PSR) before entering the interrupt service routine located in memory locations 6 and onwards (see Fig. 6.10). We also assume that the following new instructions are available:

SPS (store PSR in a memory location)
LPS (load PSR from a memory location)
ENABLE interrupt, and
DISABLE interrupt.

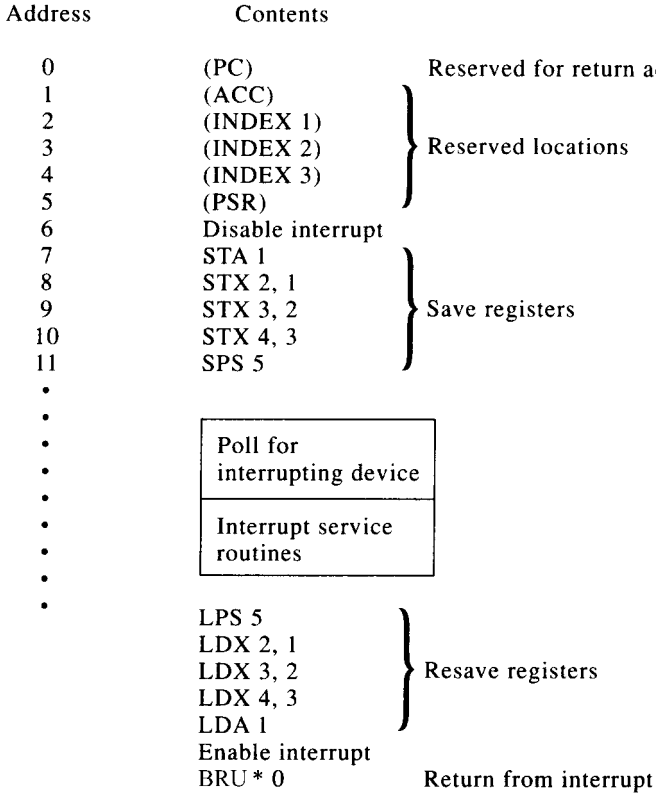



**Figure 6.10** Interrupt software (memory map)

Recall that ASC has used only 16 of the 32 possible opcodes. Any of the remaining 16 opcodes can be used for these new instructions.

The fetch sequence now looks like the following:

$T_1$: IF INT = 1 THEN MAR ← 0 ELSE MAR ← PC, READ.
$T_2$: IF INT = 1 THEN MBR ← PC, WRITE ELSE PC ← PC + 1.
$T_3$: IF INT = 1 THEN PC ← 6 ELSE IR ← MBR.
$T_4$: IF INT = 1 THEN STATE ← F ELSE (as before).

If INT is high, one machine cycle is used for entering the interrupt service routine (i.e., stores PC in location 0; sets PC = 6). The first part of the service routine (in Fig. 6.10) stores all the registers. The devices are then *polled*, as discussed later in this section, to find the interrupting device. The interrupt is serviced and the registers are restored before returning from the interrupt.

This interrupt handling scheme requires that the INT line be at 1 during the fetch cycle. That is, although the INT line can go to 1 any time, it has to stay at 1 until the end of the next fetch cycle in order to be recognized by the CPU. Further, it must go to 0 at the end of $T_4$. Otherwise, another fetch cycle in the interrupt mode is invoked. This timing requirement on the INT line can be simplified by including an *interrupt-enable* flip-flop and an *interrupt flip-flop* (INTF) into the control unit and gating the INT line into INTF at $T_1$, as shown in Fig. 6.11. The fetch sequence to accommodate these changes is shown here:

$T_1$: IF INTF = 1 THEN MAR ← 0 ELSE MAR ← PC, READ.
$T_2$: IF INTF = 1 THEN MBR ← PC, WRITE ELSE PC ← PC + 1.
$T_3$: IF INTF = 1 THEN PC ← 7 ELSE IR ← MBR.
$T_4$: IF INTF = 1 THEN STATE ← F, DISABLE INTE, RESET INTF, ACKNOWLEDGE ELSE (as before).

Note that the interrupt sequence now disables interrupts in $T_4$ thereby not requiring the DISABLE interrupt instruction at location 6 in Fig. 6.10. The interrupt service routine execution then starts at location 7. An ENABLE interrupt instruction is required. This instruction sets the interrupt-enable flip-flop and allows further interrupt. Note also that an interrupt acknowledge signal is generated by the CPU during $T_4$ to indicate to the external devices that the interrupt has been recognized. In this scheme,

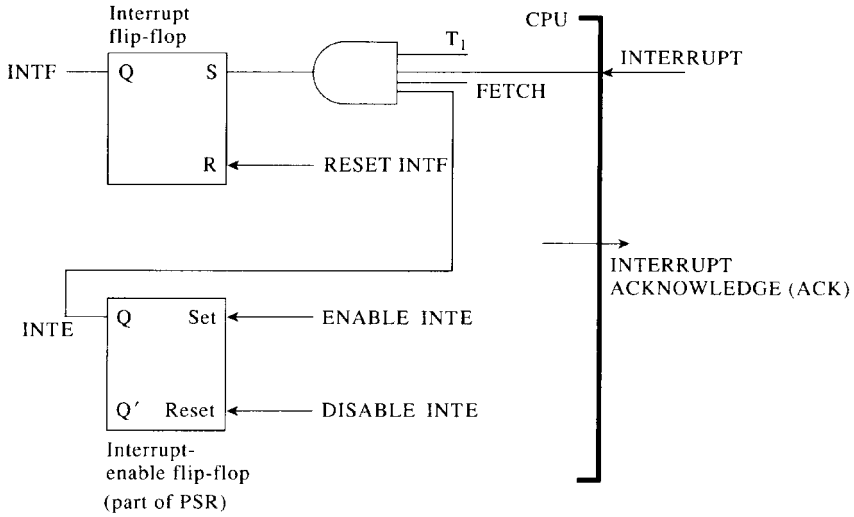



**Figure 6.11** Interrupt hardware

the interrupt line must be held high until the next fetch cycle (i.e., one instruction cycle at the worst case) for the interrupt to be recognized. Once an interrupt is recognized, no further interrupts are allowed unless interrupts are enabled.

### 6.3.2 Multiple Interrupts

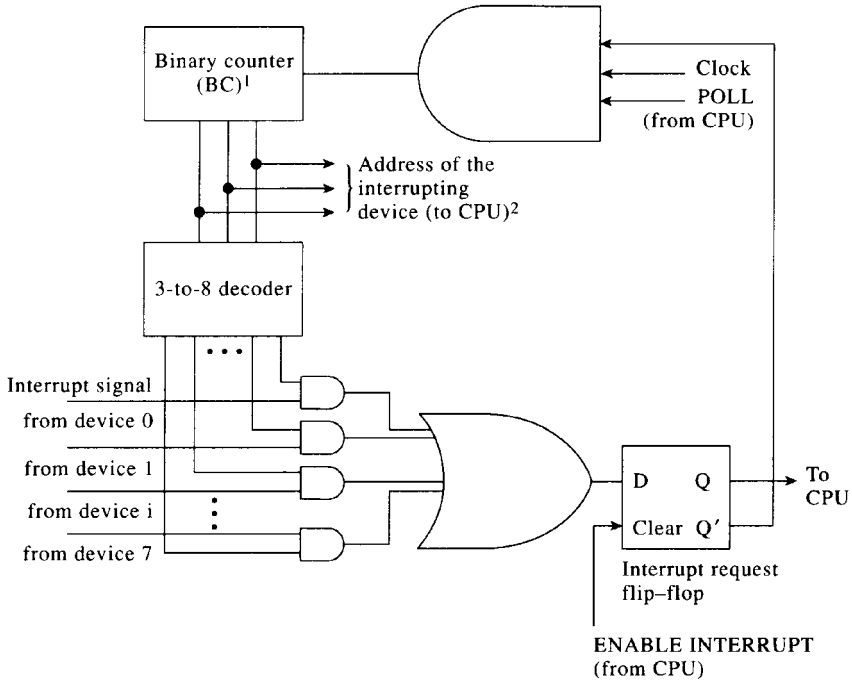
In the previous interrupt scheme, if another interrupt occurs during the interrupt service, it is completely ignored by the processor. In practice, other interrupts do occur during the interrupt service and should be serviced based on their priorities relative to the priority of the interrupt being serviced. The interrupt service routine of Fig. 6.10 can be altered to accommodate this by inserting an ENABLE interrupt instruction in location 12 right after saving registers and a DISABLE interrupt instruction just before the block of instructions that resave the registers. Although this change recognizes interrupts during interrupt service, note that if an interrupt occurs during the service, memory locations 0–5 will be over-written, thus corrupting the processor status information required for a normal return from the first interrupt.

If the processor status is saved on a stack rather than in dedicated memory locations, as done in the previous scheme, multiple interrupts can be serviced. At the first interrupt, status 1 is pushed onto the stack; at the second interrupt, status 2 is pushed onto the top of the stack. When the second interrupt service is complete, status 2 is popped from the stack, thus leaving status 1 on stack intact for the return from the first interrupt. A stack thus allows the "nesting" of interrupts. (Refer to Appendix D for details on stack implementation.)

### 6.3.3 Polling

Once an interrupt is recognized, the CPU must invoke the appropriate interrupt service routine for each interrupt condition. In an interrupt-mode I/O scheme, CPU polls each device to identify the interrupting device. Polling can be implemented in either software or hardware. In the software implementation, the polling routine addresses each I/O device in turn and reads its status. If the status indicates an interrupting condition, the service routine corresponding to that device is executed. Polling thus incorporates a priority among devices since the highest priority device is addressed first followed by lower priority devices.

Figure 6.12 shows a hardware polling scheme. The binary counter contains the address of the first device initially and counts up at each clock pulse if the CPU is in the polling mode. When the count reaches the




Notes: [1]BC starts counting when POLL = 1 and the interrupt is enabled.
[2]Interrupting device ($D_1$), sets INT request flip-flop, and BC is stopped at i (i = 0 through 7). CPU receives i as an address.

**Figure 6.12** Polling hardware (eight devices)

address of the interrupting device, the interrupt request flip-flop is set, thereby preventing the clock from incrementing the binary counter. The address of the interrupting device is then in the binary counter.

In practice, a *priority encoder* is used to connect device interrupt lines to the CPU, thereby requiring only one clock pulse to detect the interrupting device. Examples of this scheme are given later in this chapter.

### 6.3.4 Vectored Interrupts

An alternative to polling as a means of recognizing the interrupting device is to use a vectored interrupt structure like the one shown in Fig. 6.13. Here, the CPU generates an acknowledge (ACK) signal in response to an interrupt. If a device is interrupting (that is, its interrupt flip-flop is set), the ACK flip-flop in the device interface is set by the ACK signal and the device sends

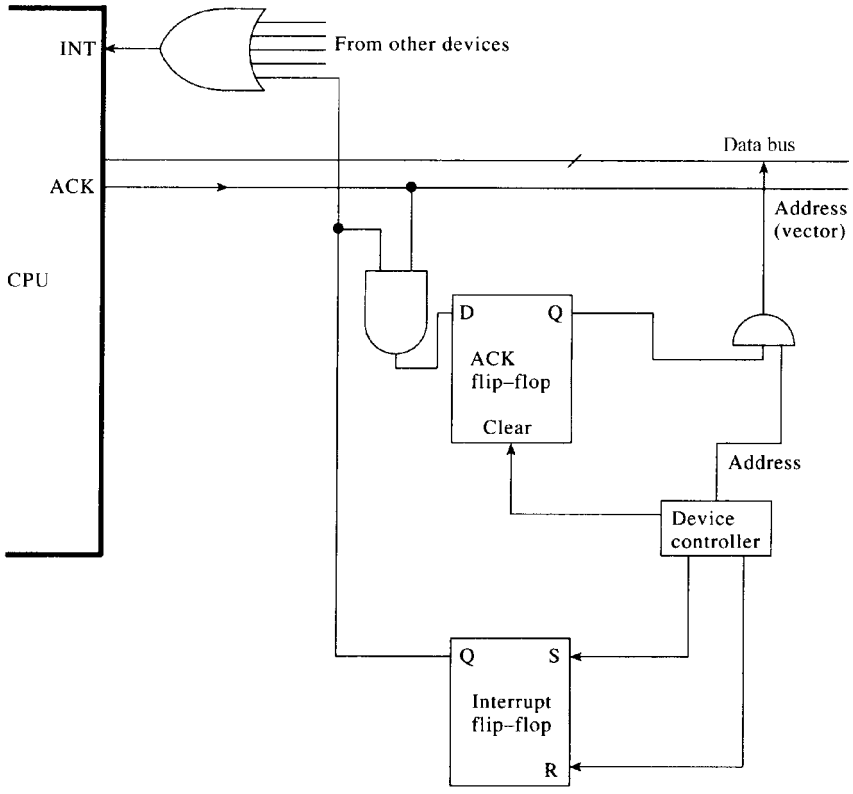
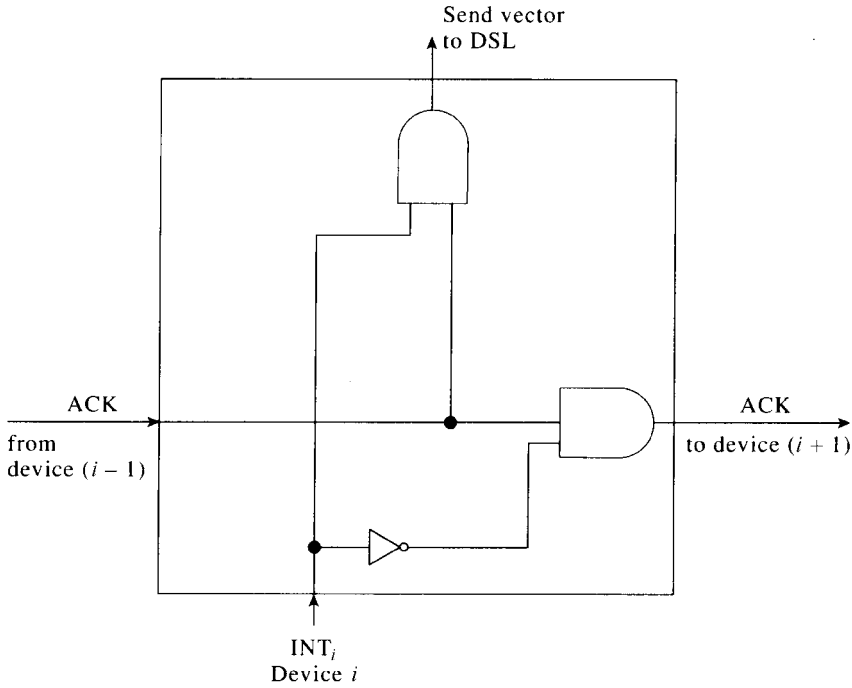



**Figure 6.13** Vectored interrupt structure

a vector onto the data bus. The CPU then reads this vector to identify the interrupting device. The vector is either the device address or the address of the memory location where the interrupt service routine for that device begins. Once the device is identified, the CPU sends a second ACK addressed to that device, at which time the device resets its ACK and interrupt flip-flops (circuitry for the second ACK is not shown in Fig. 6.13). In the structure of Fig. 6.13, all the interrupting devices send their vectors onto the data bus simultaneously in response to the first ACK. Thus the CPU can not distinguish between the devices. To prevent this and to isolate the vector of a single interrupting device, the I/O devices are connected in a *daisy chain* structure in which the highest priority device receives the ACK first, followed by the lower priority devices. The first interrupting device in the chain inhibits the further propagation of the ACK signal. Figure 6.14 shows a typical daisy chain interface. Since the daisy chain is in the ACK path, this




Note: Device ($i - 1$) has a higher priority than device ($i$); etc.

**Figure 6.14** Daisy chain interface

structure is called the *backward daisy chain*. In this scheme, the interrupt signals of all the I/O devices are ORed together and input to the CPU. Thus, any device can interrupt the CPU at any time. The CPU has to process the interrupt in order to determine the priority of the interrupting device compared to the interrupt it is processing at that time (if any). If it is processing a higher priority interrupt, the CPU will not generate the ACK.

To eliminate the overhead of comparing the priorities of interrupts, when the CPU is processing an interrupt, a *forward daisy chain* can be used in which the higher priority device prevents the interrupt signals of the lower priority device from reaching the CPU. Figure 6.15 shows a forward daisy chain. Here, each device interface is assumed to have an interrupt status flip-flop in addition to the usual interrupt and ACK flip-flops. The interrupt status flip-flop will be set when the interrupt service for that device begins, and it is reset at the end of the interrupt service. Therefore, as long as the interrupt flip-flop of the device is set (that is, as long as the device is inter-

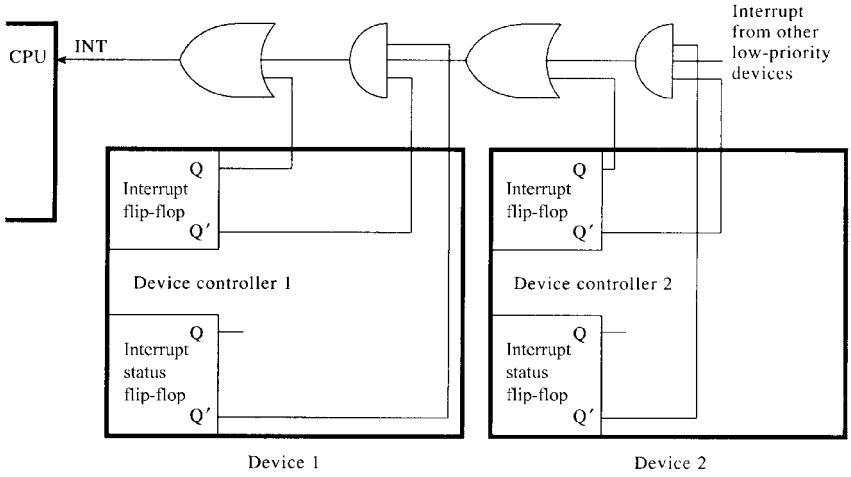



**Figure 6.15** Forward daisy chain

rupting) or the interrupt status flip-flop is set (that is, the device is being serviced), the interrupt signal from the lower priority device is prevented from reaching the CPU.

### 6.3.5 Types of Interrupt Structures

In the interrupt structure shown in Fig. 6.13, interrupt lines of all the devices are ORed together and hence any device can interrupt the CPU. This is called a *single-priority* structure because all devices are equal (in importance) as far as interrupting the CPU is concerned. Single-priority structures can adopt either polling or vectoring to identify the interrupting device. A *single-priority polled* structure is the least complex interrupt structure, since polling is usually done by software, and the slowest because of polling. In a *single-priority vectored* structure, the CPU sends out an ACK signal in response to an interrupt, and the interrupting device returns a vector that is used by the CPU to execute the appropriate service routine. This structure operates faster than the polled structure but requires a more complex device controller.

The forward daisy chain structure of Fig. 6.15 is a *multipriority* structure, since interrupting the CPU depends on the priority of the device. The highest priority device can always interrupt the CPU in this structure. A higher priority device can interrupt the CPU while the CPU is servicing an interrupt from a lower priority device. An interrupt from a lower priority device is prohibited from reaching the CPU when it is servicing a higher

priority device. Once an interrupt is recognized by the CPU, the recognition of the interrupting device is done either by polling or by using vectors. The *multipriority vectored* structure is the fastest and most complex of the interrupt structures. Table 6.1 summarizes the characteristics of these interrupt structures.

In actual systems more than one interrupt input lines will be provided at the CPU so that a hardware multilevel structure is possible. Within each level, devices can be daisy chained and each level assigned a priority. The priorities may also be dynamically changed (changed during the system operation). Figure 6.16 shows a masking scheme for such dynamic priority operations. The MASK register is set by the CPU to represent the levels that are permitted to interrupt (levels 2 and 3 are masked out and levels 1 and 4 are enabled). An INT signal is generated only if the enabled levels interrupt; that is, the levels that are masked out cannot interrupt the CPU. Note that in this scheme a device can be connected to more than one level ($D_1$ is connected to levels 1 and 2).

Each level in Fig. 6.16 receives its own ACK signal from the CPU. The ACK circuitry is not shown.

## 6.4 DIRECT MEMORY ACCESS

The programmed and interrupt mode I/O structures transfer data from the device into or out of a CPU register (accumulator in ASC). If the amount of data to be transferred is large, these schemes would overload the CPU. Data are normally required to be in the memory, especially when voluminous, and some complex computations are to be performed on them. A *direct memory access* (DMA) scheme enables a device controller to transfer data directly into or from main memory. The majority of data transfer control operations are now performed by a device controller. CPU initiates the transfer by

**Table 6.1** Characteristics of Interrupt Structures

| Structure | Response time | Complexity |
|---|---|---|
| Single-priority | | |
| Polled | Slowest | Lowest |
| Vectored | Fast | Medium |
| Multipriority | | |
| Polled | Slow | Low |
| Vectored | Fastest | Highest |

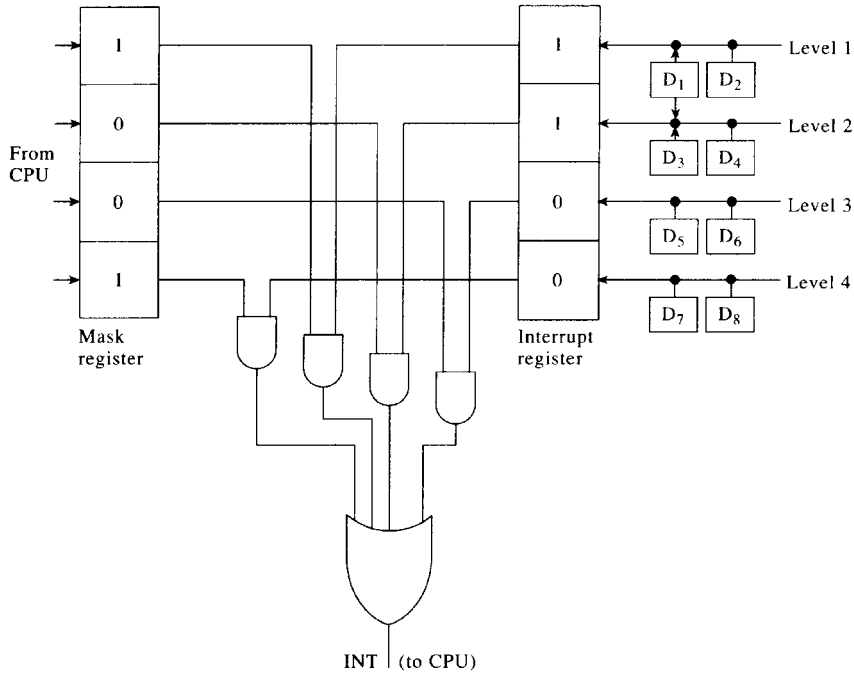
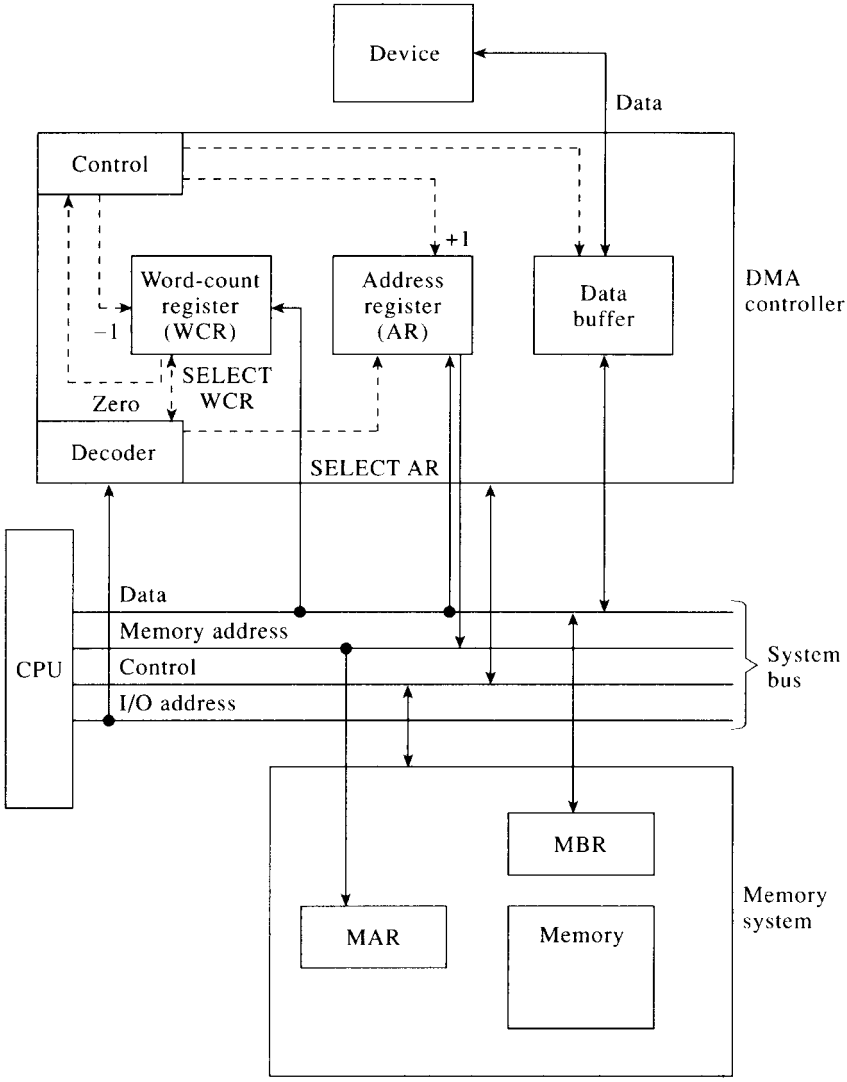



**Figure 6.16** Masking interrupts

commanding the DMA device to transfer the data and then continues with its processing activities. The DMA device performs the data transfer and interrupts the CPU only when it is completed.

Figure 6.17 shows a DMA transfer structure. The DMA device (either a DMA *controller* or a DMA *channel*) is a limited-capability processor. It will have a word-count register, an address register, and a data buffer. To start a transfer, the CPU initializes the address register (AR) of the DMA channel with the memory address from (or to) which the data must be transferred and the word-count register (WCR) with the number of units of data (words, or bytes) to be transferred. Note that the data bus is connected to these two registers. Usually, these registers are addressed by the CPU as output devices using the address bus; the initial values are transferred into them via the data bus. The DMA controller can decrement WCR and increment AR for each word transferred. Assuming an input transfer, the DMA controller starts the input device and acquires the data in its buffer register. This word is then transferred into the memory location addressed by AR; that is,




Note: Memory and I/O ADDRESS buses are shown separately for clarity; a single-address bus with a control line indicating memory or I/O operation is an alternative.

**Figure 6.17** DMA transfer structure

MAR ← AR.
MBR ← Data buffer, WRITE MEMORY.

These transfers are done using the address and data buses.

In this scheme, the CPU and the DMA device controller both try to access the memory through MAR and MBR. Since the memory cannot be simultaneously accessed both by the DMA and the CPU, a priority scheme is used to prohibit CPU from accessing the memory during a DMA operation. That is, a memory cycle is assigned to the DMA device for the transfer of data during which the CPU is prevented from accessing memory. This is called *cycle stealing*, since the DMA device "steals" a memory cycle from the CPU when it is required to access the memory. Once the transfer is complete, the CPU can access the memory. The DMA controller decrements the WCR and increments AR in preparation for the next transfer. When WCR reaches 0, a transfer-complete interrupt is sent to the CPU by the DMA controller. Figure 6.18 shows the sequence of events during a DMA transfer.

DMA devices always have higher priority than the CPU for memory access because the data available in the device buffer may be lost if not

| CPU | DMA controller |
|---|---|
| 1. Initializes AR, WCR; commands DMA controller to start data transmission; continues processing. | |
| | 2. If WCR ≠ 0, gathers data; holds CPU (i.e., "steals" a memory cycle); if WCR = 0, sends transfer-complete interrupt to CPU. |
| 3. CPU continues with any processing that does not need a memory access; if memory access is needed, tries to acquire a memory cycle if DMA is not accessing the memory. | |
| | 4. DMA transfers data; releases the memory; decrements WCR; increments AR; goes to step 2. |

**Figure 6.18** DMA transfer

transferred immediately. Hence, if an I/O device connected via DMA is fast enough, it can steal several consecutive memory cycles, thus "holding" back the CPU from accessing the memory for several cycles. If not, the CPU will access the memory in between each DMA transfer cycle.

DMA controllers can be either dedicated to one device or shared among several input/output devices. Figure 6.19 shows a bus structure that enables such a sharing. This bus structure is called *compatible I/O bus structure* because an I/O device is configured to perform both programmed and DMA transfers. DMA channels are shared by all the I/O devices. The I/O device may be transferring data at any time through either programmed or DMA paths. Some computer systems use a multiple-bus structure in which some I/O devices are connected to the DMA bus while others communicate with the CPU in a programmed I/O mode.

Refer to Section 6.10 for a complete description of the I/O structure of a modern processor (Motorola 68000).

## 6.5 BUS ARCHITECTURE

The importance of the bus architecture is as great as that of the processor, for few systems can operate without frequent data transfers along the buses in the system. As described earlier in this chapter, the system bus is responsible for interfacing the processor with the memory and disk systems, as well as other I/O devices. In addition, the bus system provides the system clock,

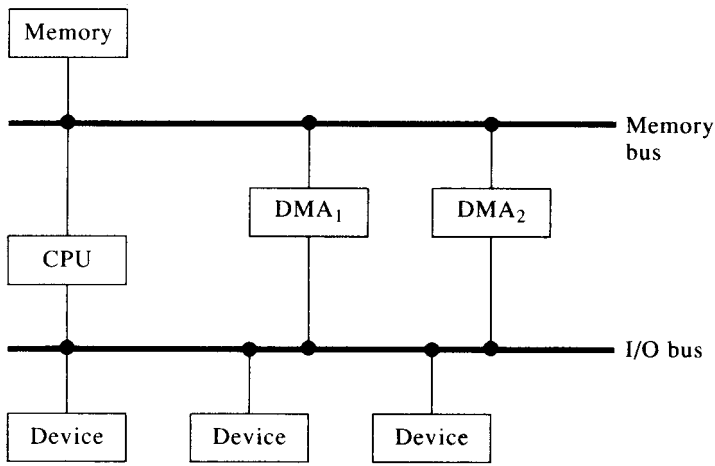



**Figure 6.19** Compatible I/O bus structure

and handles arbitration for attached devices, allowing configuration of more advanced I/O transfer mechanisms. Several standard bus architectures have evolved over the years. Section 6.10 provides relevant details of three standard bus architectures. We will now generalize the bus control and arbitration concepts.

### 6.5.1 Bus Control (Arbitration)

When two or more devices capable of becoming bus masters share a bus, a bus controller is needed to resolve the contention for the bus. Typically, in a CPU and input/output device structure, the CPU handles the bus control function. In modern computer system structures, it is common to see more than one processor connected to the bus. (A DMA structure is one such example.) One of these processors can handle the bus control function, or a separate bus controller may be included in the system structure. Figure 6.20 shows three common *bus arbitration* techniques. Here, the bus busy signal indicates that the bus has already been assigned to one of the devices as a master. The bus controller assigns the bus to a new requesting device, if the bus is not busy and if the priority of the requesting device is higher than that of the current bus master or other requesting devices. Bus request and bus grant control lines are used for this purpose. These signals are analogous to interrupt and ACK signals, respectively.

In Fig. 6.20(a), the devices are daisy chained along the bus grant path, with the highest-priority device being Device 1 and the lowest-priority device being Device $N$. Any of the devices can send a bus request to the controller. When the bus is not busy, the controller sends a grant along the daisy chain. The highest priority device among the requesting devices sets the bus busy, stops the bus grant signal from going further in the daisy chain and becomes the bus master. In (b), in response to the bus request from one or more devices, the controller polls them (in a predesigned priority order) and selects the highest priority device among them and grants the bus to it. That is, the bus grant is seen by only the selected device. In (c), there are independent bus request and grant lines. The controller resolves the priorities among the requesting devices and sends a grant to the highest-priority device among them.

## 6.6 CHANNELS

A channel is a more sophisticated I/O controller than a DMA device but performs I/O transfers in a DMA mode. It is a limited-capability processor

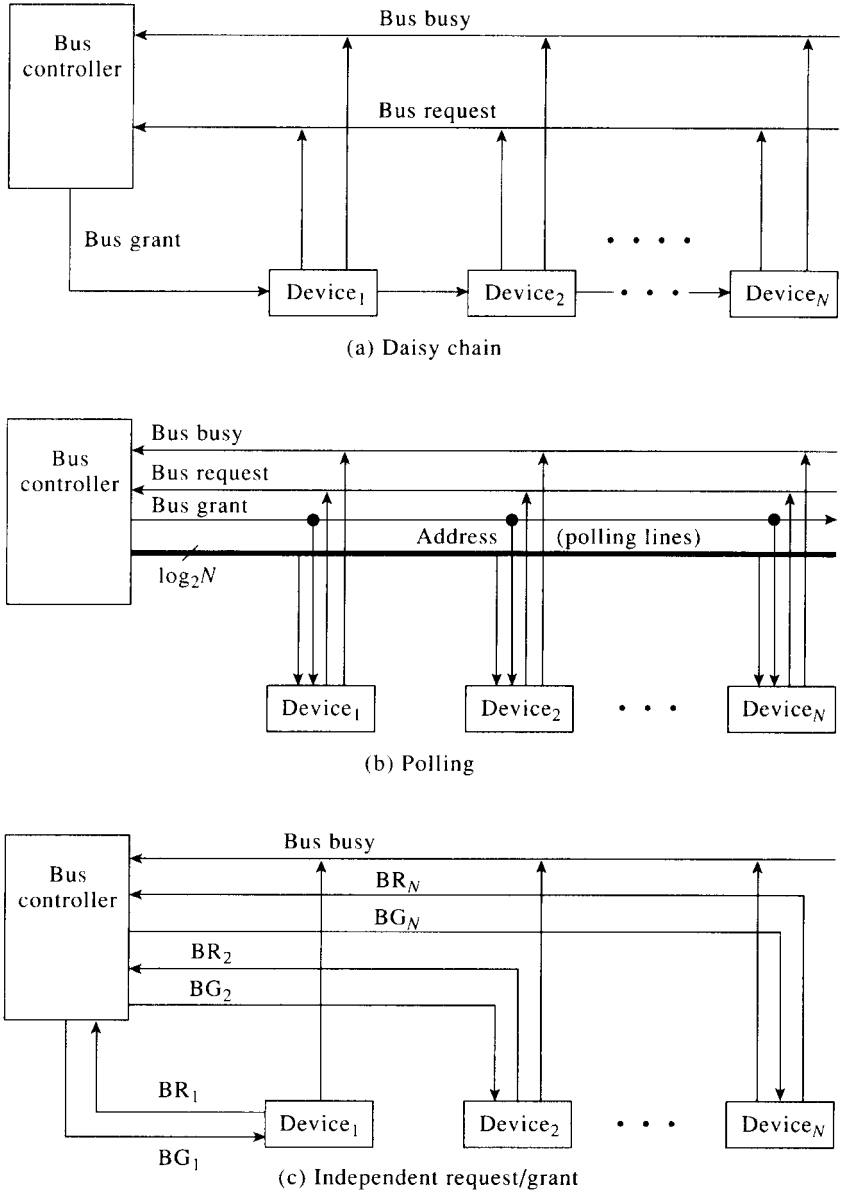



**Figure 6.20** Bus arbitration techniques

that can perform all operations a DMA system can. In addition, a channel is configured to interface several I/O devices to the memory, while a DMA controller is usually connected to one device. A channel performs extensive error detection and correction, data formatting, and code conversion. Unlike DMA, the channel can interrupt CPU under any error condition. There are two types of channels: *multiplexer* and *selector*.

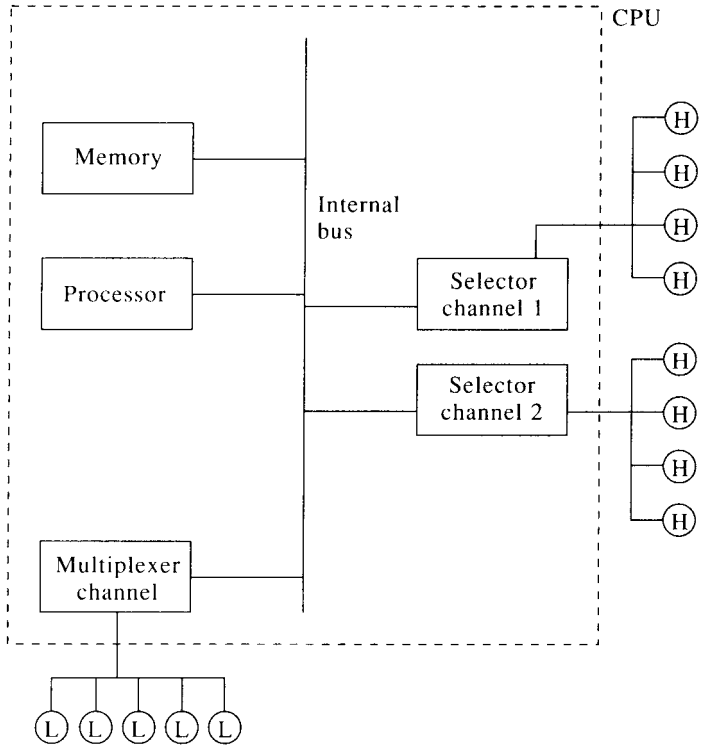
A *multiplexer channel* is connected to several low- and medium-speed devices (card readers, paper tape readers, etc.). The channel scans these devices in turn and collects data into a buffer. Each unit of data may be tagged by the channel to indicate the device it came from (in an input mode). When these data are transferred into the memory, the tags are used to identify the memory buffer areas reserved for each device. A multiplexer channel handles all the operations needed for such transfers from multiple devices after it has been initialized by the CPU. It interrupts the CPU when the transfer is complete. Two types of multiplexer channels are common: (1) character multiplexers, which transfer one character (usually one byte) from each device, and (2) block multiplexers, which transfer a block of data from each device connected to them.

A *selector channel* interfaces high-speed devices such as magnetic tapes and disks to the memory. These devices can keep a channel busy because of their high data-transfer rates. Although several devices are connected to each selector channel, the channel stays with one device until the data transfer from that device is complete.

Figure 6.21 shows a typical computer system structure with several channels. Each device is assigned to one channel. It is possible to connect a device to more than one channel through a multichannel switching interface. Channels are normally treated as a part of the CPU in conventional computer architecture; that is, channels are CPU-resident I/O processors.

## 6.7 I/O PROCESSORS (IOP)

Channels and interrupt structures perform the majority of I/O operations and control, thus freeing the central processor for internal data processing. This enhances the throughput of the computer system. A further step in this direction of distributing the I/O processing functions to peripherals is to make channels more versatile, like full-fledged processors. Such I/O processors are called *peripheral* or *front-end processors* (FEP). With the advent of microprocessors and the availability of less expensive hardware devices, it is now possible to make the front-end processor versatile enough while keeping the cost low. A large-scale computer system uses several minicomputers as FEPs while a minicomputer might use another

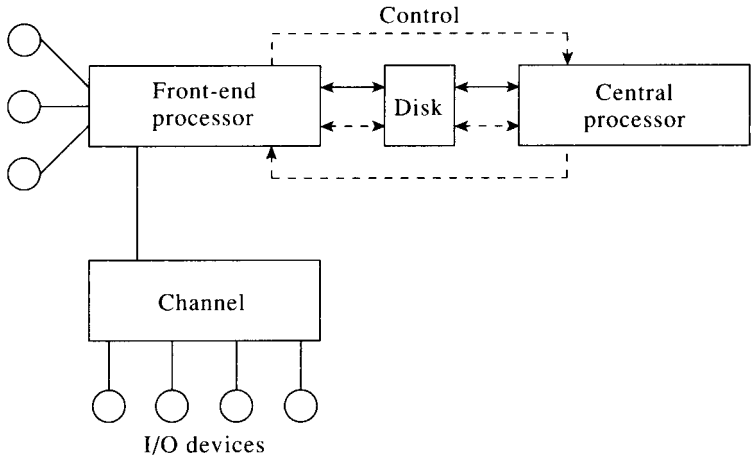



Note: H: high speed device: L: low speed device

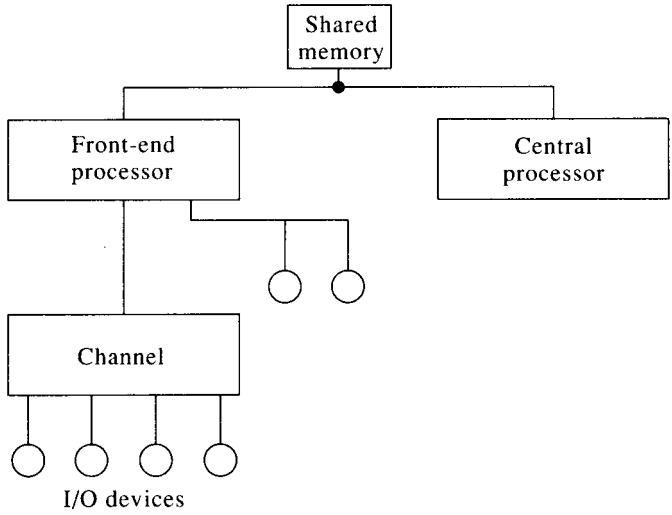
**Figure 6.21** A typical computer system

mini or a microcomputer for the FEP. Since FEPs are programmable, they serve as flexible I/O device controllers, to enable easier interfacing of a variety of I/O devices to the CPU. They also perform as much processing on the data as possible (source processing) before data are transferred into the memory. If the FEP has a writable control store (i.e., the control ROM is field-programmable), the microprogram can be changed to reflect the device interface needed.

The coupling between the FEP and the central processor is either through a disk system or through the shared memory (Fig. 6.22). In a *disk-coupled* system, the FEP stores data on the disk unit, which in turn are processed by the central processor. During the output, the central processor stores data on the disk and provides the required control information




(a) Disk-coupled system




(b) Shared-memory system

**Figure 6.22** CPU/IOP interconnection

to the FEP to enable data output. This system is easier than the shared-memory system to implement even when the two processors (FEP and CPU) are not identical because timing and control aspects of the processor-disk interface are essentially independent.

In the shared-memory system, each processor acts as a DMA device with respect to the shared memory. Hence, a complex handshake is needed, especially when the two processors are not identical. This system will generally be faster, however, since an intermediate direct-access device is not used. Figure 6.22 shows the two FEP–CPU coupling schemes.

### 6.7.1 IOP Organization

Figure 6.23 shows the organization of a typical shared-memory CPU/IOP interface. The shared main memory stores the CPU and IOP programs and contains a CPU/IOP communication area. This communication area is used for passing information between the two processors in the form of messages. The CPU places I/O-oriented information in the communication area. This information consists of the device addresses, the memory buffer addresses for data transfers, types and modes of transfer, address of the IOP program, and so on. This information is placed into the communication area by the CPU using a set of *command words* (CW). In addition to these CWs, the communication area contains the space for IOP STATUS.

While initiating an I/O transfer, the CPU first checks the status of the IOP to make sure that the IOP is available. It then places the CWs into the communication area and commands the IOP to start through the START I/O signal. The IOP gathers the I/O parameters, executes the appropriate IOP program, and transfers the data between the devices. If the data transfer involves the memory, a DMA mode of transfer is used by acquiring the memory bus using the bus arbitration protocol. Once the transfer is complete, the IOP sends a transfer-complete interrupt to the CPU through the I/O DONE line.

The CPU typically has three I/O-oriented instructions to handle the IOP: TEST I/O, START I/O and STOP I/O, each of which will be a command word. The instruction set of an IOP consists of data transfer instructions of the type: READ (or WRITE) $n$ units from (or to) device $X$ to (or from) memory buffer starting at location $Z$. In addition, the IOP instruction set may contain address manipulation instructions and a limited set of IOP program control instructions. Depending on the devices handled, there may be device specific instructions such as rewind tape, print line, seek disk address, etc.

Section 6.10 provides some details on selected commercial IOPs.

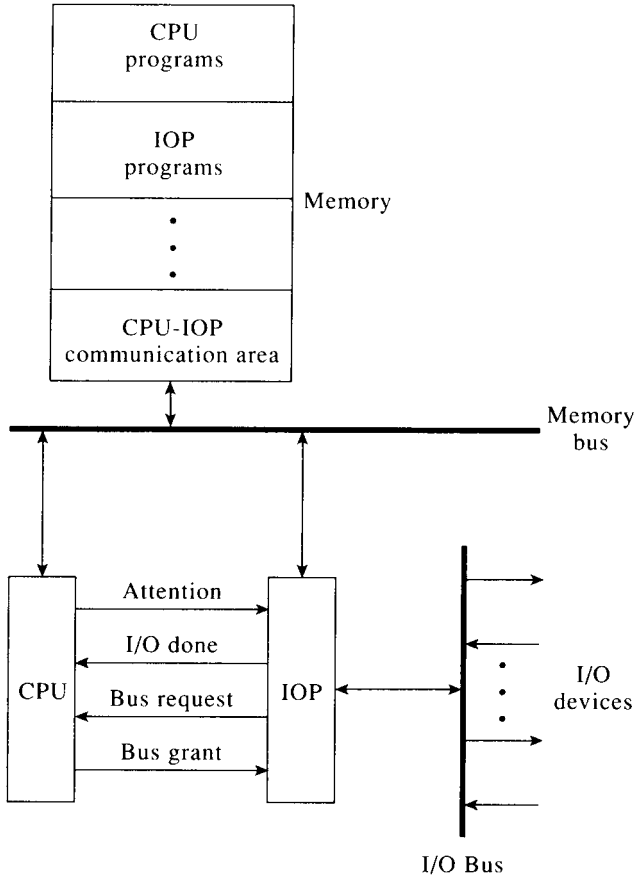



**Figure 6.23** Typical CPU-IOP interface

## 6.8 SERIAL I/O

We assumed a parallel data bus transferring a unit of data (word or byte) between the CPU, memory, and the I/O device in the earlier sections of this chapter. Devices such as low-data-rate terminals use a serial mode of data transfer containing a single line in either direction of transfer. The transmission usually is in the form of an asynchronous stream of characters. That is, there is no fixed time interval between the two adjacent characters. Figure 6.24 shows the format of an 8-bit character for asynchronous serial transmission. The transmission line is assumed to stay at 1 during the idle state, in which there is no character being transmitted. A transition from 1 to 0

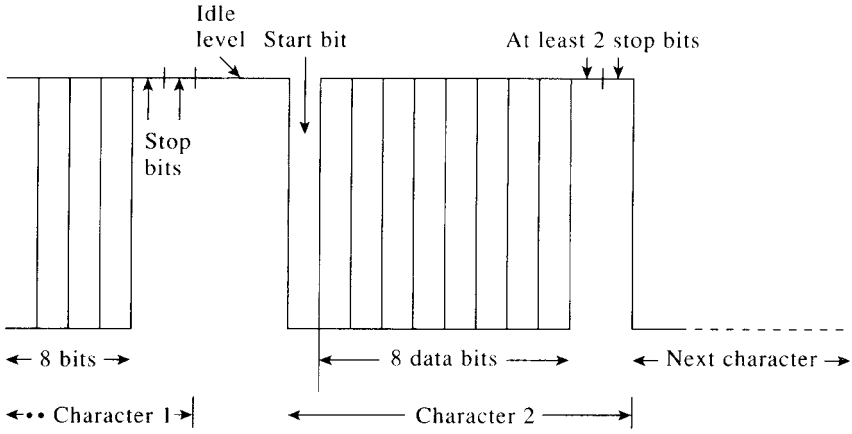



**Figure 6.24** Serial data format

indicates the beginning of transmission (*start* bit). The start bit is followed by 8 data bits, and at least 2 *stop* bits terminate the character. Each bit is allotted a fixed time interval, and hence the receiver can decode the character pattern into the proper 8-bit character code.

Figure 6.25 shows a serial transmission controller. The CPU transfers data to be output into the interface buffer. The interface controller generates the start bit, shift the data in the buffer 1 bit at a time onto output lines, and

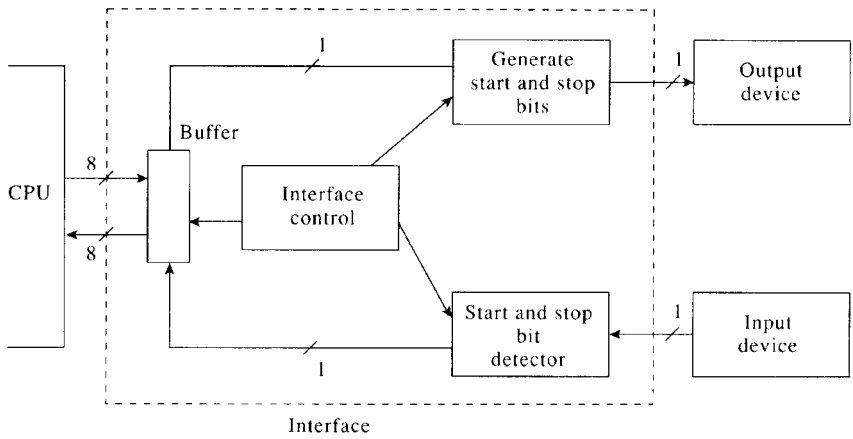



**Figure 6.25** Serial data controller

terminates with 2 stop bits. During an input, the start and stop bits are removed from the input stream and the data bits are shifted into the buffer and in turn transferred into the CPU. The electrical characteristics of this asynchronous serial interface have been standardized by the Electronic Industries Association, and it is called EIA RS232C.

The data transfer rate is measured in bits per second. For a 10-character-persecond transmission, for example, (that is, 10 characters/second × 11 bits/character, assuming 2 stop bits/character), the time interval for each bit in the character is 9.09 ms.

Digital pulses or signal levels are used in the communication of data between local devices with the CPU. As the distance between the communicating devices increases, digital pulses get distorted due to the line capacitance, noise, and other phenomena, and after a certain distance can not be recognized as 0s and 1s. Telephone lines are usually used for transmission of data over long distances. In this transmission, a signal of appropriate frequency is chosen as the *carrier*. The carrier is *modulated* to produce two distinct signals corresponding to the binary values 0 and 1. Figure 6.26 shows a typical communication structure, Here the digital signals produced by the source computer are *frequency modulated* by the modulator–demodulator (modem) into analog signals consisting of two frequencies. These analog signals are converted into their digital counterparts by the modem at the destination. If an ordinary "voice-grade" telephone line is used as the transmission medium, the two frequencies are 1170 and 2125 Hz (as used by the Bell 108 modem). More sophisticated modulation schemes are available today that allow data rates of over 20,000 bits/s.

Figure 6.27 shows the interface between a typical terminal and the CPU. The video display (screen) and the keyboard are controlled by the controller. The controller can be a simple interface that can perform basic data transfer functions (in which case the terminal is called a dumb terminal) or a microcomputer system capable of performing local processing (in which

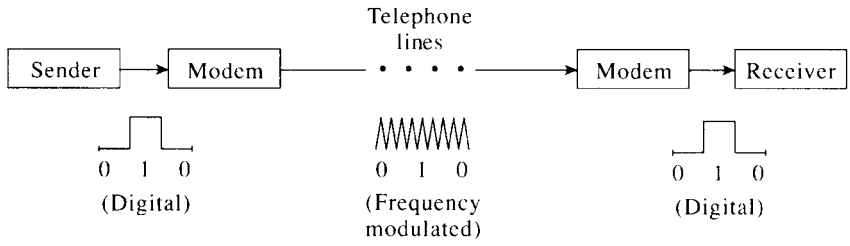



**Figure 6.26** Long-distance transmission of digital signals

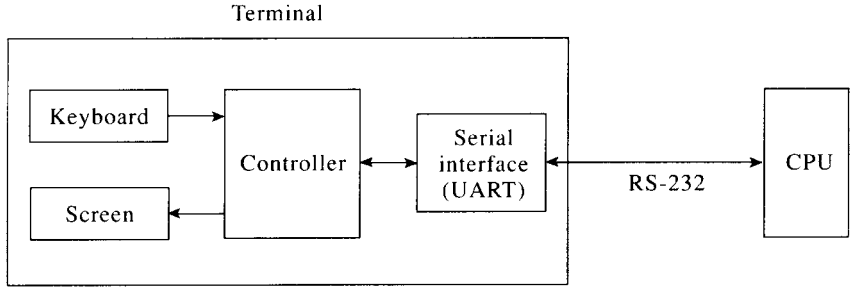



**Figure 6.27** Typical terminal/CPU interface

case the terminal is an intelligent terminal). The terminal communicates with the CPU usually in a serial mode. A *universal asynchronous receiver transmitter* (UART) facilitates the serial communication using the RS-232 interface.

If the CPU is located at a remote site, a modem is used to interface the UART to the telephone lines, as shown in Fig. 6.28. Another modem at the CPU converts the analog telephone line signals to the corresponding digital signals for the CPU.

Since the terminals are slow devices, a multiplexer is typically used to connect the set of terminals at the remote site to the CPU, as shown in Fig. 6.29.

Figure 6.30 shows the structure of a system with several remote terminal sites. The terminal multiplexer of Fig. 6.29 at each site is now replaced by a cluster controller. The cluster controller is capable of performing some processing and other control tasks, such as priority allocation and so on. The cluster controllers in turn are interfaced to the CPU through a front-end processor.

The communication between the terminals and the CPU in networks of the type shown in Fig. 6.30 follows one of the several standard protocols such as *binary synchronous control* (BSC), synchronous data link control (SDLC), and *high-level data link control* (HDLC). These protocols provide the rules for initiating and terminating transmission, handling error condi-

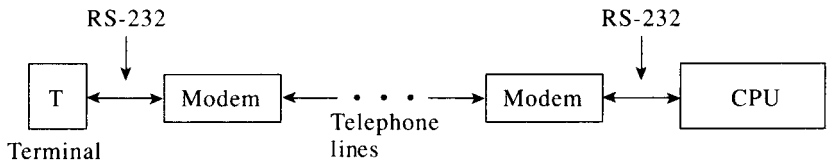



**Figure 6.28** Remote terminal interface to CPU

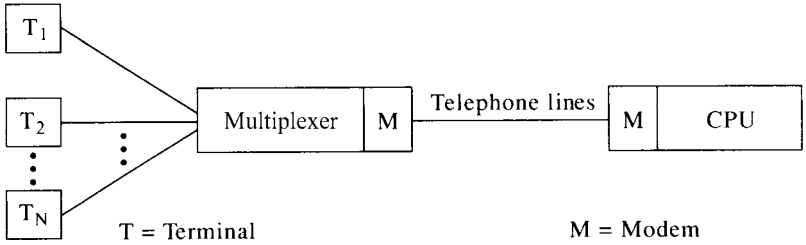



**Figure 6.29** Multiplexed transmission

tions, and data framing (rules for identifying serial bit patterns into characters, character streams into messages, and so on).

## 6.9 COMMON I/O DEVICES

A variety of I/O devices are used to communicate with computers. They can be broadly classified into the following categories:

1. Online devices such as terminals, that communicate with the processor in an interactive mode
2. Off-line devices such as printers, that communicate with the processor in a noninteractive mode

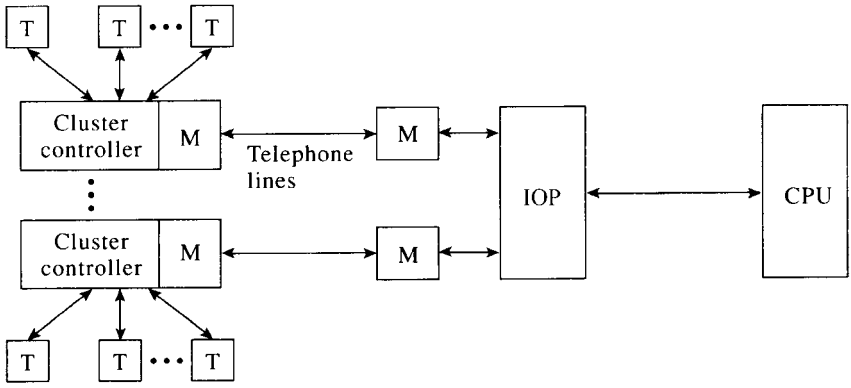



**Figure 6.30** Terminal-based computer network

3. Devices that help in real-time data acquisition and transmission (analog-to-digital and digital-to-analog converters)
4. Storage devices such as tapes and disks that can also be classified as I/O devices.

We will provide very brief descriptions of selected devices in this section. This area of computer systems technology also changes very rapidly and newer and more versatile devices are announced on a daily basis. As such, the most up-to-date information on their characteristics is only available from vendors literature and magazines such as the ones listed in the References section.

### 6.9.1 Terminals

A typical terminal consists of a monitor, a keyboard and a mouse. A wide variety of terminals are now available.

The most common monitor is still the cathode ray tube (CRT), which provides either a monochrome or color display. Flat panel displays are very common with laptop computers. These displays use liquid crystal display (LCD) technology. Displays can be either character-mapped or bit-mapped. Character-mapped monitors typically treat the display as a matrix of characters (bytes), while the bit-mapped monitors treat the display as an array of picture elements (pixels) that can be on or off, with each pixel depicting one bit of informatioan. Display technology is experiencing rapid change with newer capabilities added to displays almost daily. Sophisticated alphanumeric and graphic displays are now commonly available.

A variety of keyboards are now available. A typical keyboard used with personal computers consists of 102 keys. Various ergonomic keyboards are now appearing.

The typical mouse has three buttons. It allows pointing to any area on the screen by its movement on the mousepad. The buttons are used to perform various operations based on the information at the selected spot on the screen.

In addition to these three components of a typical terminal, various devices such as light pen, joystick, microphones (for direct audio input), speakers (for audio output) are very commonly found in terminals today.

### 6.9.2 Printers

The cheapest type of printers are dot-matrix printers. The print head in these printers contains from 7 to 24 needles that can be activated electromagnetically to form each character in a line printed. Inkjet printers are very

common now a days. They use dot matrix technology except that each needle is now an ink nozzle. There are several types of laser printers now on the market. These use the same technology as photocopy machines and offer excellent image quality, flexibility, moderate costs and very good speed. Laser printers offer mainly monochrome capability, while the other technologies offer both monochrome and color printing.

### 6.9.3 Scanners

Scanners are used to transfer the content of a hard copy document to the machine memory. The image can be retained as is and transmitted as needed. It can also be processed (reduce, enlarge, rotate, etc.). If the image is to be treated as a word processable document, optical character recognition is first performed after scanning it. Scanners of various capabilities and price ranges are now available.

### 6.9.4 A/D and D/A Converters

Figure 6.31 shows a digital processor controlling an analog device. Real-time and process-control environments, monitoring laboratory instruments, fall into the application mode shown in the figure. Here the analog signal produced by the device is converted into a digital bit pattern by the A/D converter. The processor outputs the data in digital form, which is converted into analog form by the D/A converter. D/A converters are resistor-ladder networks that convert the input $n$-bit digital information into the corresponding analog voltage level.

A/D converters normally use a counter along with a D/A converter. The contents of the counter are incremented, converted into analog signals, and compared to the incoming analog signal. When the counter value corresponds to a voltage equivalent to the input analog voltage, the counter is stopped from incrementing further. The contents of the counter, then, correspond to the equivalent digital signal.

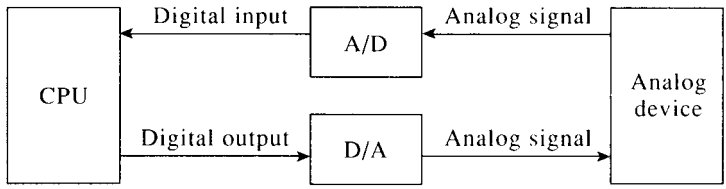



**Figure 6.31** Interfacing an analog device

### 6.9.5 Tapes and Disks

Storage devices such as magnetic tapes (reel-to-reel, cassette and streaming cartridge) and disks (hard and floppy, magnetic and optical) described in Chapter 3, are also used as I/O devices.

Table 6.2 lists typical data transfer rates offered by some I/O devices, mainly to show the relative speeds of these devices. The speed and capabilities of these devices change so rapidly that such tables become outdated even before they are published. The description provided in this section is intentionally very brief. The reader should refer to vendors' literature for current information on these and other devices.

## 6.10 EXAMPLE I/O STRUCTURES

Computer systems are generally configured around a single or multiple bus structure. There is no general baseline structure for commercial systems; each system is unique. But most of the modern systems utilize one or more standard bus architectures to allow easier interface of devices from disparate vendors. This section provides brief descriptions of the following selected bus standards and I/O structures.

1. Motorola 68000 (a complete I/O structure example)
2. VME, Multibus I and II (bus standards)
3. Intel 8089 (a simple I/O processor of historical interest)
4. Intel 21285 (a versatile interconnect processor)
5. Digital Equipment Corporation PDP-11 (an I/O structure of historical interest)
6. Control Data 6600 (another I/O structure of historical interest).

**Table 6.2** Data transfer rates of I/O devices

| Device | Transfer rate |
|---|---|
| Display terminals | 10–240 characters/second |
| Impact printers | 100–3000 lines/minute |
| Nonimpact printers | 100–40,000 lines/minute |
| A/D and D/A converters | 15–300 $\times$ $10^6$ samples/second |
| Magnetic tape | 15,000–30,000 characters/second |
| Cassette tape | 10–400 characters/second |
| Hard disk | 10–25 $\times$ $10^6$ bits/second |
| Floppy disk | 25,000 characters/second |

Almost all of these systems have now been replaced by higher-performance, more versatile versions. Nevertheless, these structures depict the pertinent characteristics more simply. Refer to manufacturers' manuals for details on the latest versions.

### 6.10.1 Motorola 68000

Although MC68000 has been replaced by higher-performance counterparts, it is included here to provide a complete view of the I/O structure of a commercially available processor system. The MC68000 is a popular microprocessor with a 32-bit internal architecture that is capable of interfacing with both 8- and 16-bit-oriented peripherals. Figure 6.32 shows the functional grouping of signals available on the processor. A brief description of these signals follows.

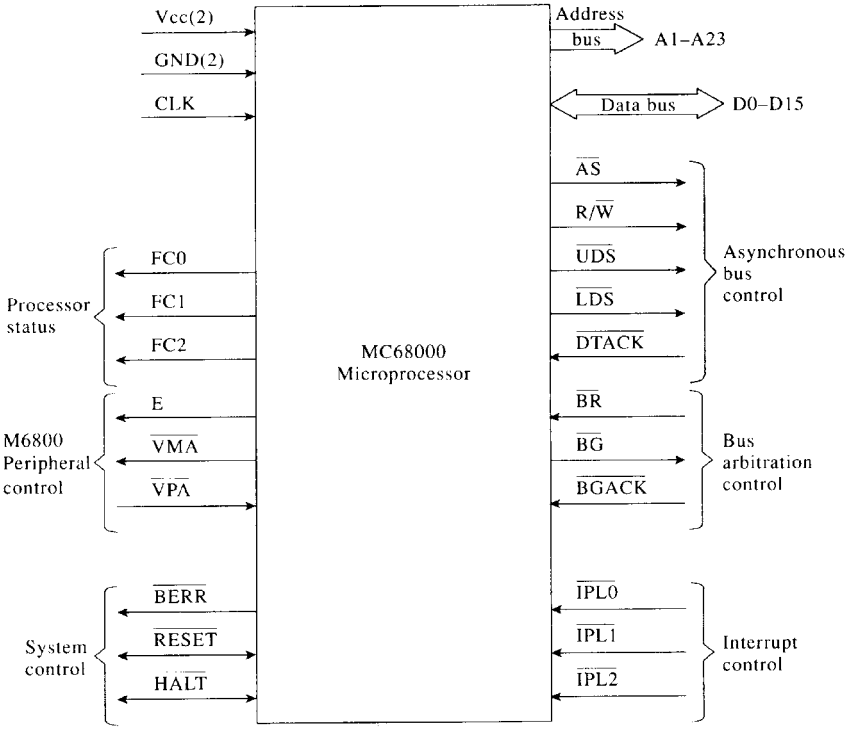



**Figure 6.32** Motorola 68000 processor (Courtesy of Motorola Inc.)

The MC68000 memory space is byte addressed. Address bus lines (A1–A23) always carry an even address corresponding to a 16-bit (2-byte) word. The upper data strobe (UDS) and lower data strobe (LDS), if active, refer to the upper or lower byte respectively of the memory word selected by A1–A23. D0–D15 constitute the 16-bit bidirectional data bus. The direction of the data transfer is indicated by the READ/WRITE (R/W) control signal. The ADDRESS STROBE (AS) signal, if active, indicates that the address (as determined by the address bus, UDS, and LDS) is valid. DATA ACKNOWLEDGE (DTACK) is used to indicate that the data have been accepted (during a write operation) or that the data are ready on the data bus (during a read). The processor waits for the DTACK during the I/O operations with variable speed peripherals. There are three bus arbitration signals: BUS REQUEST (BR), BUS GRANT (BG), and BUS GRANT ACKNOWLEDGE (BGAC). There are three interrupt lines (IPL0–IPL2) that allow an interrupting device to indicate one of the seven levels of interrupt possible: level 1 being the lowest priority, level 7 the highest, and level 0 meaning no interrupt. The processor communicates with a 8-bit-oriented peripherals of MC6800 through the ENABLE (E), VALID MEMORY ADDRESS (VMA) and VALID PERIPHERAL ADDRESS (VPA) control signals. The communication through these lines is synchronous. The FUNCTION CODE (FC0–FC2) output signals indicate the type of bus activity currently undertaken (such as interrupt acknowledge, supervisor/user program/data memory access, etc.) by the processor. There are three system control signals: BUS ERROR (BERR), RESET, and HALT.

## Programmed I/O on MC68000

The MC68000 uses memory-mapped I/O, since no dedicated I/O instructions are available. The I/O devices can be interfaced either to the asynchronous I/O lines or to the MC68000-oriented synchronous I/O lines. We will now illustrate interfacing an input and an output port to MC68000 to operate in the programmed I/O mode through the asynchronous control lines, using the parallel interface/timer chip MC68230.

The MC 68230 (See Fig. 6.33) is a peripheral interface/timer (PI/T) consisting of two independent sections: the ports and the timer. In the port section there are two 8-bit ports (PA0-7 and PB0-7), four handshake signals (H1–H4), two general I/O pins (I/O), and six dual-function pins. The dual-function pins can individually work either as a third port (C) or an alternate function related to either port A or B or the timer. Pins H1–H4 are used in various modes: to control data transfer to and from ports, as general-purpose I/O pins, or as interrupt-generating inputs with corresponding vectors.

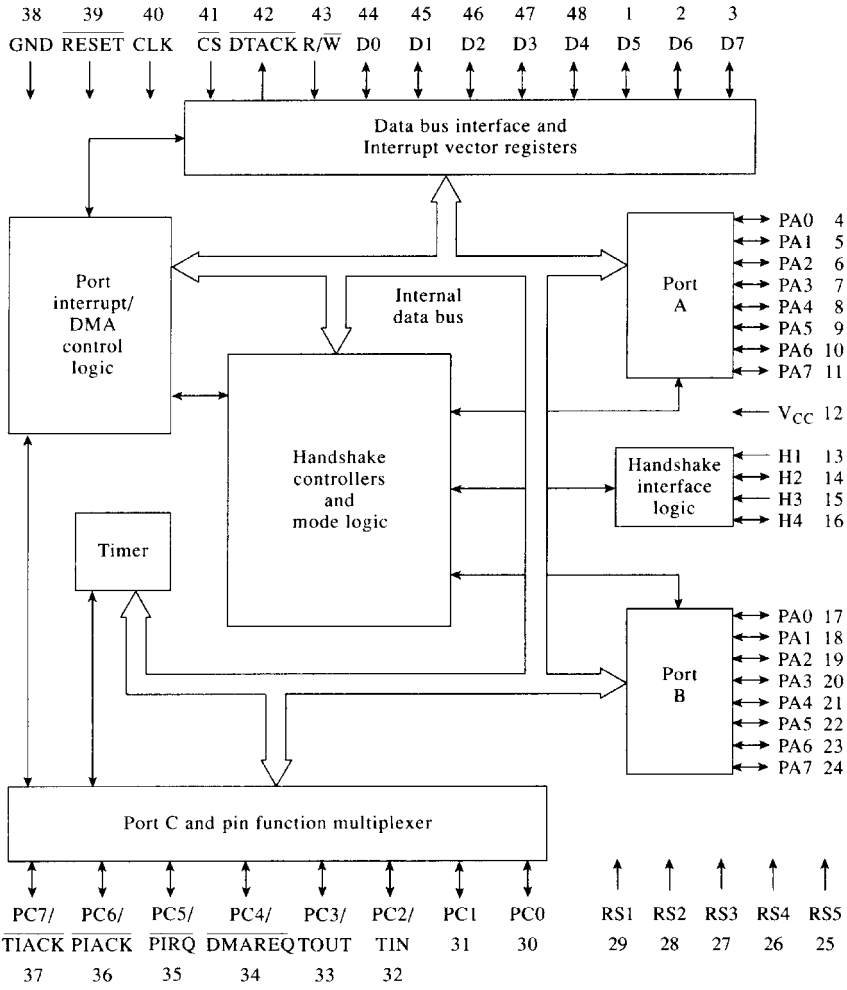



**Figure 6.33** MC68230 functional diagram (Courtesy of Motorola Inc.)

The timer consists of a 24-bit counter and a prescaler. The timer I/O pins (TIN, TOUT, $\overline{\text{TIACK}}$) also serve as port C pins.

The system data bus is connected to pins D0–D7. Asynchronous transfer between the MC68000 and PI/T is facilitated by data transfer acknowledge ($\overline{\text{DTACK}}$), register selects (RS1–RS5), timer interrupt acknowledge ($\overline{\text{TIACK}}$), read/write (R/$\overline{\text{W}}$) and port interrupt acknowledge ($\overline{\text{PIACK}}$).

We will restrict this example to the ports section. The PI/T has 23 internal registers, each of which can be addressed using RS1–RS5. Associated with each of the three ports is a data register, a control register, and a data direction register (DDR). Register 0 is the port general control register (PGCR) that controls all the ports. Bits 6 and 7 of this register are used to configure the ports in one of the four possible modes as shown in the following table. The other bits of the PGCR are used for input/output handshaking operations.

| Bit 7 | Bit 6 | Mode |
|---|---|---|
| 0 | 0 | Unidirectional (8-bit) |
| 0 | 1 | Unidirectional (16-bit) |
| 1 | 0 | Bidirectional (8-bit) |
| 1 | 1 | Bidirectional (16-bit) |

When the ports are configured in unidirectional mode, corresponding control registers are used to further define the submode of operation. For example, in the unidirectional 8-bit mode, bit 7 of a control register being 1 signifies a bit-oriented input/output in the sense that each bit of the corresponding port can be programmed independently. To configure a bit as an output or an input bit, the corresponding DDR bit is set to 1 or 0 respectively. DDR settings are ignored in the bidirectional transfer mode.

Figure 6.34 shows the MC68000 interface with MC68230 in which the following addresses are used:

| Register | Address (Hex) |
|---|---|
| PGCR | 100001 |
| PADDR | 100003 |
| PBDDR | 100007 |
| PACR | 10000D |
| PBCR | 10000F |
| PADR | 100011 |
| PBDR | 100013 |

Figure 6.35 shows an assembly language program that reads an 8-bit input from port A and writes the data to port B.

## MC68000 Interrupt System

The MC68000 interrupt system is divided into two types: *internal* and *external*. The internal interrupts are called *exceptions* and correspond to condi-

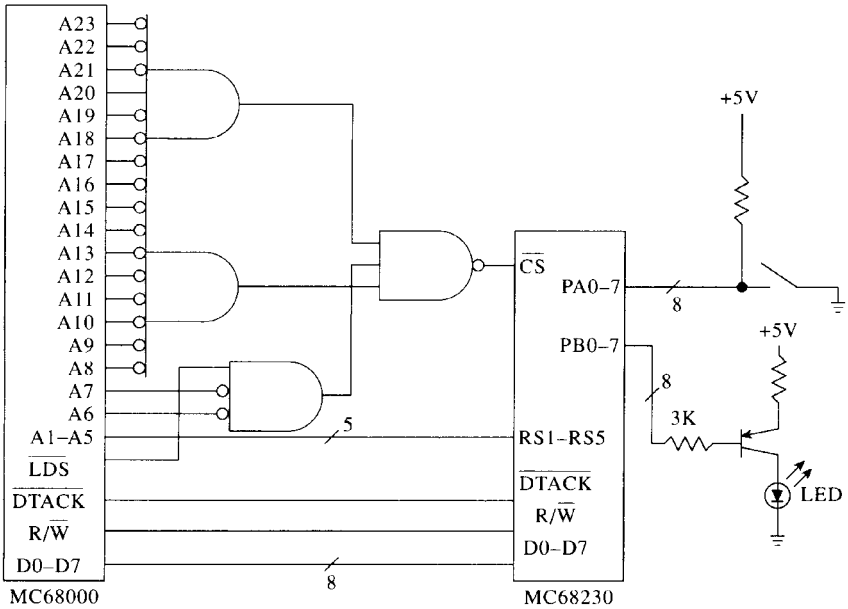



**Figure 6.34** Interfacing I/O ports to MC68000

tions such as divide by zero, illegal instruction, and user-defined interrupts by the use of trap instructions. The external interrupts correspond to the seven levels of interrupts brought through the $\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$ lines, as mentioned earlier: level 0 indicates no interrupt, level 7 is the highest priority interrupt and is non-maskable by the processor, and the other levels are recognized by the processor if the processor is in a lower-priority processing mode.

A four-byte field is reserved for each possible interrupt (external and internal). This field contains the beginning address of the interrupt service routine corresponding to that interrupt. The beginning address of the field is the vector corresponding to that interrupt. When the processor is ready to service an interrupt, it pushes the program counter and the status register (SR) onto the stack, updates the priority mask bits in SR, puts out the priority level of the current interrupt on A1–A3, and sets FC0–FC2 to 111 to indicate the interrupt acknowledge ($\overline{\text{IACK}}$). In response to the $\overline{\text{IACK}}$, the external device either can send an 8-bit vector number (non-auto vector) or can be configured to request the processor to generate the vector automatically (autovector). Once the vector (v) is known, the processor jumps to the location pointed to by the memory address (4v). The last

```
; ASSIGN ADDRESSES

PGCR      EQU       $100001   ;  $ indicates hex
PADDR     EQU       $100003
PBDDR     EQU       $100007
PACR      EQU       $10000D
PBCR      EQU       $10000F
PADR      EQU       $100011
PBDR      EQU       $100013

; INITIALIZE  ALL REGISTERS
;  MOVE.B   Source, Destination    Moves bytes
; #  indicates immediate address mode

          MOVE.B    #$00,PGCR ;Select unidirectional
                              ; 8-bit mode
          MOVE.B    #$FF,PACR ;Bit input/output mode for port A
          MOVE.B    #$FF,PBCR ;Bit input/output mode for port B
          MOVE.B    #$00,PADDR;All bits of A are input
          MOVE.B    #$FF,PBDDR;All bits of B are output

;READ AND WRITE

START:    MOVE.B    PADR,D1   ; Input from A into processor
                              ; register D1

          MOVE.B    D1,PBDR   ;Output to B
          JMP       START     ; Repeat
```

**Figure 6.35** MC68000 assembly language program for input/output

instruction in the service routine has to be a return from interrupt (RTE), which pops the program counter (PC) and SR from the stack, thus returning to the former state.

Figure 6.36 shows the vector map. Memory addresses 00H (i.e., hexadecimal 00) through 2FH contain vectors for conditions such as reset, but error, trace, divide by zero, and so on. There are seven autovectors. The operand of the TRAP instruction indicates one of the possible fifteen trap conditions, thus generating one of the fifteen trap vectors. Vector addresses 40H through FFH are reserve for user nonautovectors. Spurious interrupt vector handles the interrupt due to noisy conditions.

In response to the $\overline{\text{IACK}}$, if the external device asserts $\overline{\text{VPA}}$, the processor generates one of the seven vectors (19H through 1FH) automatically. Thus, no external hardware is needed to proved the interrupt vector.

For nonautovector mode, the interrupting device places a vector number (40H through FFH) on the data but lines D0–D7 and asserts $\overline{\text{DTACK}}$. The processor reads the vector and jumps to the appropriate service routine.

| Vector address | | Vector number |
|---|---|---|
| 00H to 2FH | Reset, Bus Error, etc. | 00H to 16H |
| 30H to 5CH | Unassigned | 0CH to 17H |
| 60H, 62H | Spurious interrupt | 18H |
| 64H, 66H | Autovector 1 | 19H |
| 68H, 6AH | Autovector 2 | 1AH |
| 6CH, 6EH | Autovector 3 | 1BH |
| 70H, 72H | Autovector 4 | 1CH |
| 74H, 76H | Autovector 5 | 1DH |
| 78H, 7AH | Autovector 6 | 1EH |
| 7CH, 7EH | Autovector 7 | 1FH |
| 80H to BCH | TRAP instructions | 20H to 2FH |
| COH<br>⋮<br>FCH | Unassigned | 30H<br>⋮<br>3FH |
| 100H<br>⋮<br>3FCH | User interrupts (Nonautovector) | 40H<br>⋮<br>FFH |

**Figure 6.36** MC68000 vector map

Due to system noise, it is possible for the processor to be interrupted. In that case, on the receipt of IACK, an external timer can activate the $\overline{\text{BERR}}$ (after a certain time). When the processor receives the $\overline{\text{BERR}}$ in response to $\overline{\text{IACK}}$, it generates the spurious interrupt vector (18H).

Figure 6.37 shows the hardware needed to interface two external devices to the MC68000. The priority encoder (74LS148) generates the three-bit interrupt signal for the processor based on the relative priorities of the devices connected to it. When the processor recognizes the interrupt, it puts the priority level on A1–A3, sets F0–F2 to 111, and activates $\overline{\text{AS}}$. The 3-to-8 decoder (74LS138) thus generates the appropriate $\overline{\text{IACK}}$. When device 1 is interrupting, the $\overline{\text{VPA}}$ line is activated in response to $\overline{\text{IACK1}}$, thus activating an autovector mode. For device 2, $\overline{\text{IACK2}}$ is used to gate its vector onto data lines D0–D7 and to generate $\overline{\text{DTACK}}$.

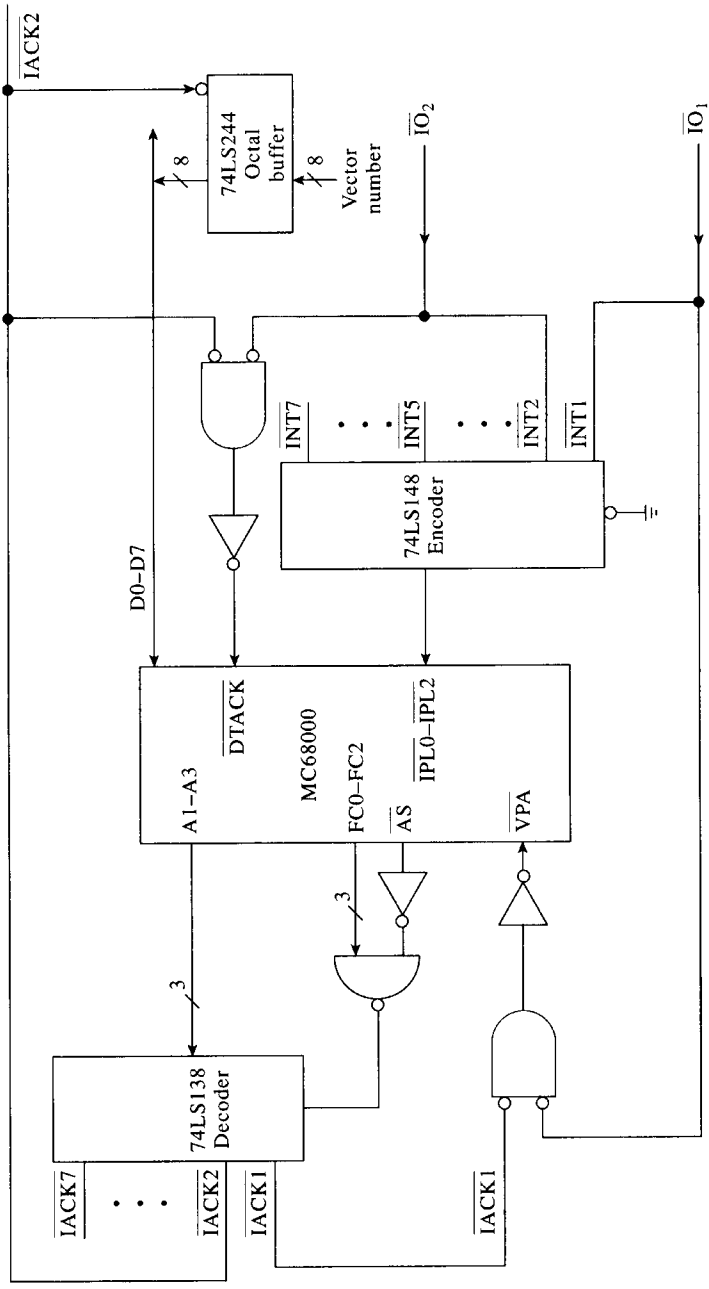



**Figure 6.37** Vectored Interrupts on MC68000

### MC68000 DMA

To perform a DMA, the external device requests the bus by activating $\overline{\text{BR}}$. One clock period after receiving $\overline{\text{BR}}$, MC68000 enables the bus grant line $\overline{\text{BG}}$ and relinquishes (tristates) the bus after completing the current instruction cycle (as indicated by the $\overline{\text{AS}}$ going high). The external device then enables $\overline{\text{BGACK}}$, indicating that the bus is in use. The processor waits for the $\overline{\text{BGACK}}$ to go high in order to use the bus.

## 6.10.2 Bus Standards

This section provides the details of three bus standards: Multibus I, Multibus II and VME bus. The description here is focused on data and control signals, arbitration mechanisms, and interrupt generation and handling. The references listed at the end of the chapter provide further details on electrical and timing issues.

### Multibus I

Although an older standard, IEEE standard 796 (Multibus I), has proved that a well-designed architecture can provide a viable system for many years. Systems utilizing this bus have existed since about 1976, but some implementations have been used in more modern systems. The lengthy existence of the Multibus architecture was a direct result of the design goals: simplicity, processor and system flexibility, ease of upgrading, and suitability for harsh environments.

A Multibus I system comprises one or more boards connected via a passive backplane. The architecture is processor agnostic, so much so that systems utilizing the bus have been designed around processors ranging from the Intel 8080 to the 386, and other families such as the Z80 and Motorola 68030. Multibus I supports both single and multiprocessor architectures. The IEEE 796 standard provides for a wide variation in implementations and this text will attempt to describe the bus in terms of its overall capability, while noting relevant allowed variations.

Multibus I control signals are active low, and terminated by pull-up resistors. Two clock signals are present. The bus clock (BCLK) runs at 10 MHz, and is used to synchronize bus contention operations. The constant clock (CCLK) is also 10 MHz, but is routed to the masters and slaves to use as the master clock. Four signals (MWTC, MRDC, IOWC, and IORC) are signals a bus master can use to initiate a read or write to memory or I/O space. During a write operation, an active signal denotes that the address carried on the address lines is valid. During a read operation, the transfer

from active to inactive indicates that the master has received the requested data from the slave. Slaves raise the transfer acknowledge (XACK) signal to indicate to a master that it has completed a requested operation. The initialize signal (INIT) can be generated to reset the system to its initial state. The lock (LOCK) signal may be used to lock the bus. This will be further explained later, during the discussion of multiprocessing features.

Multibus I supports up to 24 address lines (ADR0–ADR23). Memory sizes greater than 16 Mb (the maximum allowed with 24 address lines) could be handled by a memory management unit. Both 8 and 16 bit processors are supported, and in the case of an 8-bit system, both even and odd bytes are accessible via a swapping mechanism. In I/O access, either 8 or 16 address lines are used, giving an I/O address space of up to 64K separate from the data space.

There are 16 data lines (DAT0–DAT15), although for 8-bit systems only the first 8 are valid. DAT0 is the least significant bit. The data lines are shared between memory and I/O devices. The use of both 8- and 16-bit systems is permitted by the byte high enable signal (BHEN), which is used on 16-bit systems to signify the validity of the other 8 data lines. Two inhibit lines (INH1 and INH2) can be asserted by a slave to inhibit another slave's bus activity during a memory read or write.

Eight interrupt lines (INT0–INT7) are available, and can be configured to work in either a direct or bus-vectored interrupt scheme. The interrupt acknowledge signal (INTA) is used in the bus-vectored mechanism by a master to freeze the interrupt status and request the interrupting device to place its vector address onto the bus data lines.

Bus arbitration is handled via five bus exchange lines. The bus busy (BUSY) line is driven by the master currently in ownership of the bus to signify the state of the bus. It is a bidirectional signal driven by an open-collector gate, and synchronized by BCLK. In either serial or parallel priority schemes, other bus masters use the bus priority in (BPRN) to indicate a request for bus control along the chain or parallel circuitry. In serial mechanism, the bus priority out signal (BPRO) is passed on to the next higher bus master to propagate a request for bus control. In the parallel-priority scheme, each bus master requesting access to the bus raises the bus request signal (BREQ). The parallel circuitry resolves the priorities, and enables BPRN for the highest priority master requesting control. An optional signal, the common bus request (CBREQ) can be used by any master to signal a request for bus control. This allows a master of lower priority to request bus control.

Multibus I data transfer is asynchronous, and supports DMA transfer. Devices in the system may be either masters or slaves, with slaves such as memory unable to initiate a transfer. Multibus I supports up to 16 bus

masters, using either serial or parallel priority schemes. Data transfer takes place in the following steps:

1. The bus master places the memory or I/O address on the address lines.
2. The bus master generates the appropriate command signal.
3. The slave either accepts the data in a write, or places the data on the data lines for a read.
4. The slave sends the transfer acknowledge signal back to the master.
5. The bus master removes the signal on the command lines, and then clears the address and data lines.

Since Multibus I data transfers are asynchronous, it is possible that due to error a transfer could extend indefinitely. To prevent this, a bus timeout can be implemented to terminate the cycle after a preset interval of at least 1 ms. For any memory transfer, a slave can assert the inhibit lines to inhibit the transfer of another slave. This operation has been implemented for use in diagnostic applications and devices, and is not widely used in normal operation. When transferring between 8 and 16 bit devices, the byte high enable signal (BHEN) and the least significant address line (ADR0) are used to define whether the even or odd byte will be transferred. In an even-byte transfer, both are inactive. For the odd-byte transfer, BHEN is inactive and ADR0 is active. When transferring between two 16-bit devices, both signals are active.

Multibus I supports two methods of interrupts: direct (non-bus-vectored) and bus vectored. Direct interrupts are handled by the master without the need for the device address being placed on the address lines. Bus-vectored interrupts are handled by the master interrogating the interrupting slave instead of determining the vector address. When the interrupt request occurs, the bus master interrupts its processor, which generates the INT command and freezes the state of the interrupt logic so that the priority of the request can be analyzed. After the INT command, the bus master determines the address of the highest-priority request, and places the address on the bus address lines. Depending on the size of the address of the interrupt vector, either one or two more INTA commands must be generated. The second one causes the slave to transmit the low-order (or only) byte of its interrupt vector on the data lines. If necessary (for 16-bit addresses), the third INTA causes the high-order byte to be placed on the data lines. The bus master then uses this address to service the interrupt.

Since Multibus I can accommodate several bus masters, there must be a mechanism for the masters to negotiate for control of the bus. This can take place through either a serial or parallel priority negotiation. In the

serial negotiation, a daisy chain technique is used. The priority in and out of each master are connected to each other in order of their priority. When a bus master requests control of the bus, it generates its BPRO signal, and blocks its BPRN signal, thus locking out the requests of all the lower-priority masters. In the parallel mechanism, a bus arbiter receives the BPRN and BREQ signals of each master. The bus arbiter then determines the priority of the request and performs the request. Bus arbiters are usually not designed into the backplane.

When the I/O of the processor is bottlenecked by the bus, Multibus I allows the use of bus extensions, which take over high throughput transfers such as DMA. In a standard Multibus I system, the maximum throughput is limited to 10 Mb/s. Processors later in the life of the Multibus I architecture were capable of a much higher rate; a 20 MHz 386 chip can transfer at 40 Mb/s. Multibus I may implement both the iLBX and iSBX extensions. ILBX provides a fast memory mapped interface, in an expansion board with the same form factor as the Multibus I standard. A maximum of two masters can share the bus, which limits the need for complex bus arbitration. The arbitration that does take place has been modified to be asynchronous to data transfers. ILBX slaves are available to the system as byte addresable memory resources controlled directly from the iLBX bus lines. For 16-bit transfers, these improvements allow a maximum throughput of 19 Mb/s. The iSBX expansion board implements its own I/O and DMA transfers, taking over much of the function of the Multibus I architecture. The implementation of the iSBX bus extension marked the beginning of an evolution towards Multibus II.

## Multibus II

Multibus I was introduced in 1974 and it is fundamentally CPU and memory bus. Then it evolved to a multiple-master shared memory bus capable of solving most real-time applications of that time. But in the late 1980s the users demanded a new standard bus. Therefore Intel set up a consortium with 18 industry leaders to define the scope and possibilities of the next generation of bus – Multibus II.

The consortium decided that no single bus can be used to satisfy all user needs, therefore a multiple bus structure consisting of four subsystems was defined. The four subsystems are: the iSBX bus for incremental I/O expansions, a local CPU and memory expansion bus and two system buses – one serial and one parallel.

Consider a local area network or LAN. This system is functionally partitioned – each node of the LAN is independent of others and optimized

for a part of overall problem. This solution gives the system architect freedom to choose the hardware and software for each node that best fits the subtask. Moreover, each of the systems can be upgraded individually.

The Multibus II allows to create a "very local" network within a single chassis. Dividing a multi-CPU application into a set of networked subsystems allows optimization of each subsystem for the subtask it works on. If a subsystem is complex the resources may be spread over multiple boards and communicate via local expansion bus.

A double Eurocard format, the IEEE 1101 standard, with dual 96-pin DIN connectors was chosen for Multibus II standard. A U-shaped front panel, licensed from Siemens, Germany, was chosen for its enhanced electromagnetic interference (EMI) and radio-frequency interference (RFI) shielding properties.

The popularity of Multibus I products encouraged adoption of iSBX (IEEE 894) for incremental I/O bus. The IEEE/ANSI 1296 specification does not define the exact bus for local expansion. The bus varies depending on performance required in subsystem design. Intel initiated iLBX II standard optimized for 12-MHz Intel 80286 processor. For high-performance local expansion buses Futurebus can be used.

Because the CPU-memory bus on most buses is not adequate for system-level requirements, *system space* was defined in Multibus II specification. It consists of two parts: interconnect space and message space. Interconnect space is used for initialization, self-diagnostics and configuration requirements. In order to maintain compatibility with existing buses the traditional CPU-memory space is retained.

Intel implemented the Multibus II parallel system bus in a single VLSI called the message-passing coprocessor (MPC). This chip consists of 70,000 transistors and contains almost all logic needed to interface processor to the system bus. The parallel system bus is defined as a 32-bit bus clocking at 10 MHz, thus allowing data transfers up to 40 Mbits/s.

The serial system bus (SSB) is defined as a 2 MBit/s serial bus, but not implemented. The software interface to an SSB must be identical to that for a parallel bus.

Being a major part of IEEE/ANSI 1296 Multibus II specification, interconnect address space addresses board identification, initialization, configuration and diagnostics requirements. Interconnect space is implemented as an ordered set of 8-bit registers on long-word (32-bit) boundaries – in this way a small endian microprocessor such as the 8086 family and a big endian microprocessor such as 68000 family access the information in an identical manner. The software can use the interconnect address space to get information about the environment it operates in, the functionality of board and the slot in which the board operates.

The identification registers contain information about board type, its manufacturer and installed components. They are read-only registers. Configuration registers are read/write locations that can be set and changed by system software. Diagnostics registers are used for self-diagnosis of the board.

Interconnect space is based on the idea that it is possible to locate boards within the backplane by their physical slot positions. This principle, called geographical addressing, is used for system-wide initialization. Each board has firmware with a standardized 32 byte header format containing a 2-byte vendor ID, a 10-byte board name and other vendor-defined information. At boot time the system software scans each board to locate its resources and then loads appropriate drivers. This method eliminates the need for reconfiguration each time a new board added to the system. Each board in the system performs its own initialization and testing using the firmware and passes all information needed to the operating system which, in turn, generates a resource-location map to be used as a basis for message-passing addresses, thus achieving slot independence.

In general, a board manufacturer also supplies other function records to make additional functionality of the board accessible through interconnect space. Types of function record for common functions such as memory configuration and serial I/O are defined.

The diagnosing philosophy is standardized by Intel. Each Multibus II board should be capable of self-testing and reporting the status in interconnection space. The self-testing can be invoked during power-on initialization or explicitly from the console. If a hardware failure is detected a yellow LED on the font panel will illuminate, helping the operator easily define and replace the board.

The high-performance of Multibus II is achieved by decoupling activities between the CPU-memory local bus and the system bus. This approach gives two advantages: parallelism of operations is increased and one bus bandwidth does not limit transfer rate of another. The local bus and the system bus work independently and in parallel. This is achieved by using nine 32-byte FIFO buffers integrated into the MPC. Five of them are used for interrupts (one sends and four receive) and four for data transfer (two to send, two to receive).

The Multibus II specification introduces a hardware-recognized data type called a packet. The packets are moved on subsequent clock edges of the 10 MHz synchronous bus, thus a single packet can occupy the bus for no longer than 1 $\mu$s. An address is assigned to each board in the system. This address is used in source and destination fields. Seven different types are defined by the standard. The packets are divided into two groups: unsoli-

cited and solicited. Unsolicited packets are used for interrupts while solicited packets are used for data transfers. The data fields are user-defined and may be from 0 bytes to 32 (28 for unsolicited packets) bytes long in 4-byte increments.

Unsolicited packets are always a "surprise" for the destination-bus MPC. These packets are similar to interrupts in shared-memory systems with additional feature to carry up to 28 bytes of information. General interrupt packets can be sent between any two boards. Broadcast interrupts are sent to all boards in the system. Three other types (buffer request, reject and grant) are used initiate large data transfers.

Unlike unsolicited packets, solicited packets are not unpredictable to the destination MPC. These packets are used for large data transfers between boards. Up to 16 MB of data can be transferred by solicited packets. All operations on packets such as creation, bus arbitration, error checking and correction are done by MPC and transparent to the local processor.

The Multibus II system bus utilizes a distributed arbitration scheme. Each board has daisy-chain circuitry. The system bus is continually monitored by each of MPCs. The scheme supports two arbitration algorithms: fairness and high priority.

In the fairness mode, if the bus is being used, the MPC waits before requesting the bus; when the bus is not busy, the MPC makes the request and waits for a bus grant; once the MPC uses the bus, it does not request the bus until all other requesters have used it. If no bus requests have arrived since last usage of the bus, the MPC accesses the bus without performing an arbitration cycle. This mechanism prevents the bus from being monopolized by a single board. Since each MPC uses the bus for a maximum of 1 $\mu$s and arbitration is resolved in parallel, no clock cycles are wasted and all transfers operate back-to-back.

The high-priority mode is used for interrupts. The requesting MPC bus controller is guaranteed the next access to the bus. Usually interrupt packets are sent in high-priority mode, and this means that most interrupt packets have a maximum latency of 1 $\mu$s, although in very rare instances, when $N$ boards initiate interrupt packets within the same 1 $\mu$s window. These packets may have up to $N - 1$ $\mu$s latency.

The parallel system bus is implemented on a single 96-pin connector. The signals are divided into five groups: central control, address/data, system control, arbitration and power.

The parallel system bus is synchronous and a great care is taken to maintain a clear 10 MHz-system clock. The IEEE/ANSI 1296 specification details precisely what happens upon each of the synchronous clock edges so there is no ambiguity. Numerous state machines that track bus activity are defined in order to guarantee compatibility.

The board in slot 0, called the central services module (CSM), generates all of the central control signals. It can be implemented on a CPU board, a dedicated board, or on the backplane itself. The module drives reset (RST*)[1] to initialize the system; combinations of DCLOW* and PROT* is used to distinguish between cold start, warm start and power failure recovery. Two system clocks (BCKL* at 10 MHz and CCLK* at 20 MHz) are generated.

The IEEE/ANSI 1296 specification defines the parallel system bus as a full 32-bit (AD0*–AD31*) with parity control (PAR0*–PAR3*). Because it is defined as a multiplexed address/data bus, the system control lines are used to distinguish between data and addresses. Since all transfers are checked for parity error, in case of parity failure the MPC bus controller retries the operation. If the error is not recovered within 16 tries, MPC interrupts its host processor and asks for assistance.

There are ten system control lines (SC0*–SC9*), and their functions are also multiplexed. SC0* defines the current state of the bus cycle (request or reply) and how SC1–SC7 lines should be interpreted. SC8 provides even parity over SC4–SC7, and SC9 provides even parity over SC0–SC3. The following table from the handbook by DiGiacomo (1990) summarizes functions of the system control lines:

| Signal | Function during request phase | Function during reply phase |
|---|---|---|
| SC0* | Request phase | Reply phase |
| SC1* | Lock | Lock |
| SC2* | Data width | End-of-transfer |
| SC3* | Data width | Bus owner read |
| SC4* | Address space | Replier ready |
| SC5* | Address space | Agent status |
| SC6* | Read or write | Agent status |
| SC7* | Reserved | Agent status |
| SC8* | Even parity on SC ⟨7–4⟩* | Even parity on SC ⟨7–4⟩* |
| SC9* | Even parity on SC ⟨3–0⟩* | Even parity on SC ⟨3–0⟩* |

There is a common bus request line BREQ*. The specification defines that the distributed arbitration scheme grants the bus to the numerically higher requesting board as identified on lines ARB0–ARB5. As said earlier, the scheme supports two arbitration modes: fairness and high priority.

[1] *denotes that the line is active-low

The parallel system bus is particularly easy to interface to. An I/O replier need implement only a single *replying agent* state machine shown in Fig. 6.38. In the following example the assumption is made that requestor makes only valid requests.

The replying agent is a bus monitor. State transitions occur on the falling clock edge. The replier remains in the "wait for request" state until the start of request cycle is detected (SC0* is low). If the request is addressed to the replier (ADDR is high), then there is a state transition to a new state controlled by a local ready signal (REPRDY).

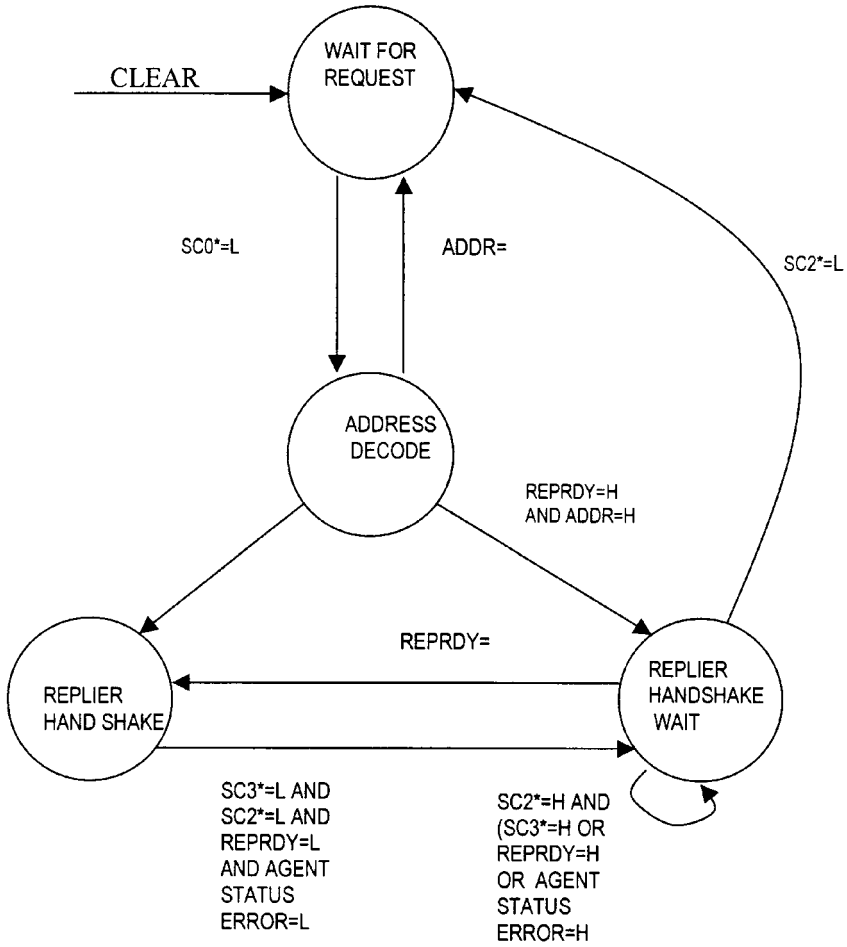



**Figure 6.38** Multibus II state diagram (Courtesy of Intel Corporation)

If REPRDY is low, then it waits until the device is ready. When ready, it waits for requestor to be ready (SC3* is low) and performs the data transfer. Then it checks if it is a multibyte transfer (SC2 is high), if it is, the state machine decides to accept or ignore the data in the remainder of the cycle. If additional data can not be handled, then the replier sends the continuation error and waits for the requestor to terminate the cycle. If the additional data can be handled, the replier oscillates between the replier wait state and replier handshake state until the last packet (SC2* is low) is received. If it is not a multibus transfer, the replier returns to the wait state.

Because of the simple standardized interface and processor independence, the Multibus II became very popular on the market. Many vendors produced Multibus II compatible boards with many different functions. Later the IEEE/ANSI 1296.2 standard was adopted and it expands the Multibus II with "live insertion" capabilities.

### VMEbus

The VMEbus is a standard backplane interface that simplifies integration of data processing, data storage, and peripheral control devices in a tightly coupled hardware configuration. The VMEbus interfacing system is defined by the VMEbus specification. This system has been designed to do the following:

1. Allow communication between devices without disturbing the internal activities of other devices interfaced to the VMEbus.
2. Specify the electrical and mechanical system characteristics required to design devices that will reliably and unambiguously communicate with other devices interfaced to the VMEbus.
3. Specify protocols that precisely define the interaction between devices.
4. Provide terminology and definitions that describe system protocol.
5. Allow a broad range of design latitude so that the designer can optimize cost and/or performance without affecting system compatibility.
6. Provide a system where performance is primarily device limited, rather than system interface limited.

The VMEbus functional structure consists of backplane interface logic, four groups of signal lines called buses, and a collection of functional modules. The lowest layer, called the backplane assess layer, is composed of the backplane interface logic, the utility bus modules, and the arbitration

bus modules. The VMEbus data transfer layer is composed of the data transfer bus and priority interrupt bus modules.

The data transfer bus allows bus masters to direct the transfer of binary data between themselves and slaves. The data transfer bus consists of 32 data lines, 32 address lines, 6 address modifier lines, and 5 control lines. There are nine basic types of data transfer bus cycles: read, write, unaligned write, block read, block write, read–modify–write, address-only, and interrupt acknowledge cycle.

The slave detects data transfer bus cycles initiated by a master, and when those cycles specify its participation, transfers data between itself and the master.

The priority interrupt bus allows interrupter modules to request interrupts from interrupt handler modules. The priority interrupt bus consists of seven interrupt request lines, one interrupt acknowledge line, and an interrupt acknowledge daisy chain.

The interrupter generates an interrupt request by driving one of the seven interrupt request lines. When its request is acknowledged by an interrupt handler, the interrupter provides 1, 2, or 4 bytes of status or identification to the interrupt handler. This allows the interrupt handler to service the interrupt.

The interrupt handler detects interrupt requests generated by interrupters and responds by asking for status or identification information.

The interrupt acknowledge daisy-chain driver's function is to activate the interrupt acknowledge daisy chain whenever an interrupt handler acknowledges an interrupt request. This daisy chain ensures that only one interrupter responds with its status or identification when more than one has generated an interrupt request.

The VMEbus is designed to support multiprocessor systems where several masters and interrupt handlers may need to use the data transfer bus at the same time. It consists of four bus request lines, four daisy-chained bus grant lines, and two other lines called bus clear and bus busy.

The requestor resides on the same board as a master or interrupt handler. It requests use of the data transfer bus whenever its master or interrupt handler needs it. The arbiter accepts bus requests from requester modules and grants control of the data transfer bus to only one requester at a time. Some arbiters have a built-in time-out feature that causes them to withdraw a bus grant if the requesting board does not start using the bus within a prescribed time. This ensures that the bus is not locked up as a result of a result of a transient edge on a request line. Other arbiters drive the bus clear line when they detect a request for the bus from a requester whose priority is higher than the one that is currently using the bus. This ensures that the response time to urgent events is bounded.

The utility bus includes signals that provide periodic timing and coordinate the power-up and power-down sequences of the VMEbus system. Three modules are defined by the utility bus: The system clock driver, the serial clock driver, and the power monitor. It consists of two-clock line, a system-reset line, an a.c. fail line, a system fail line, and a serial data line.

The system clock driver provides a fixed-frequency 16-MHz signal. The serial clock driver provides a periodic timing signal that synchronizes operation of the VMEbus. The VMEbus is part of the VMEsystem architecture and provides an interprocessor serial communication path.

The power monitor module monitors the status of the primary power source to the VMEbus system. When power strays outside the limits required for reliable system operation, it uses the a.c. fail line to broadcast a warning to all boards on the VMEbus system in time to effect graceful shutdown.

This system controller board resides in slot 1 of the VMEbus backplane and includes all the one-of-a-kind functions that have been defined by the VMEbus. These functions include the system clock driver, the arbiter, the interrupt acknowledge daisy-chain driver, and the bus timer.

Two signaling protocols are used on the VMEbus: closed-loop protocols and open-loop protocols. Closed-loop protocols use interlocked bus signals while open-loop protocols use broadcast bus signals. The address strobe and data strobes are interlocked signals that are especially important. They are interlocked with the data acknowledge or bus error signals and coordinate the transfer of addresses and data. There is no protocol for acknowledging a broadcast signal; instead, the broadcast is maintained long enough to ensure that all appropriate modules detect the signal. Broadcast signals may be activated at any time.

The smallest addressable unit of storage on the VMEbus is the byte. Masters broadcast the address over the data transfer bus at the beginning of each cycle. These addresses may consist of 16, 24, or 32 bits. The 16-bit addresses are called short addresses, the 24-bit addresses are called standard addresses, and the 32-bit addresses are called extended addresses. The master broadcasts a 6-bit address modifier (AM) code along with each address to tell slaves whether the address is short, standard, or extended.

There are four basic data transfer capabilities associated with the data transfer bus: D08(EO) (even and odd byte), D08(O) (odd byte only), D16, and D32.

Five basic types of data transfer cycles are defined by the VMEbus specification. These cycle types include the read cycle, the write cycle, the block read cycle, the block write cycle and the read–modify–write cycle.

Two more types of cycles are defined: the address-only cycle and the interrupt acknowledge cycle. With the exception of the address-only cycle, which does not transfer data, all these cycles can be used to transfer 8, 16 or 32 bits of data.

Read and write cycles can be used to read or write 1, 2, 3, or 4 bytes of data. The cycle begins when the master broadcasts an address and an address modifier mode. Block transfer cycles are used to read or write a block of 1–256 bytes of data. The VMEbus specification limits the maximum length of block transfers to 256 bytes. Read–modify–write cycles are used both to read from and write to a slave location in an indivisible manner, without permitting any other master to access that location. The VMEbus protocol allows a master to broadcast the address for the next cycle, while the data transfer for the previous cycle is still in progress. The VMEbus provides two ways that processors in a multiprocessing system can communicate with each other: by using interrupt request lines or by using location monitors.

The location monitor functional module is intended for use in multiple-processor systems. It monitors all data transfer cycles over the VMEbus and activates an on-board signal whenever an access is done to any of the locations that it is assigned to watch.

In multiple-processor systems, events that have global importance need to be broadcast to some or all of the processors. This can be accomplished if each processor board includes a location monitor.

The VMEbus provides both the performance and versatility needed to appeal to a wide range of users. Its rapid rise in popularity has made it the most popular 32-bit bus. VMEbus systems will accommodate the inevitable changes easily and without making existing equipment obsolete.

When examining and comparing bus architectures, one must look at several attributes, such as the transfer mechanism, interrupt servicing, and bus arbitration. VMEbus and Multibus I & II are excellent buses to demonstrate capabilities and responsibilities, because between them they implement a variety of different mechanisms. Interrupt servicing could take place via direct or vectored techniques. But transfers can be synchronous or asynchronous. Bus masters can implement a serial (daisy-chained) or parallel method to compete for control of the bus. By using techniques such as asynchronous arbitration and direct memory access (DMA), modern buses are able to reduce the bottleneck the processor faces when accessing memory and I/O devices. Although currently processors are advancing more quickly than the buses they attach to, this is largely a result of the need for bus standards. As faster standardized buses become prevalent, this gap may eventually disappear, or at least be reduced to a level where zero wait state execution becomes a reality.

### 6.10.3 Intel 8089 IOP

Intel 8089 is an I/O support processor for Intel 8086-based systems. It consists of two independent DMA channels. Corresponding to each channel there is an *I/O address register* (IOAR) and a *data word count register* (DC), just as in any DMA controller. In addition, there is a *program counter* (PC) and other registers that are used in executing IOP programs. The CPU views the 8089 as two independent IOPs since the two DMA channels can execute different I/O programs concurrently.

The IOP is interfaced to the 8086 system bus using a bus interface unit, as shown in Fig. 6.39. There is a 20-bit arithmetic-logic unit (ALU) that is used for address computations and an assemble/disassemble unit that is used to change a 16-bit data unit into two 8-bit units and vice versa, during the 8- and 16-bit data transfers. In Fig. 6.39, the IOP is attached to a 20-bit

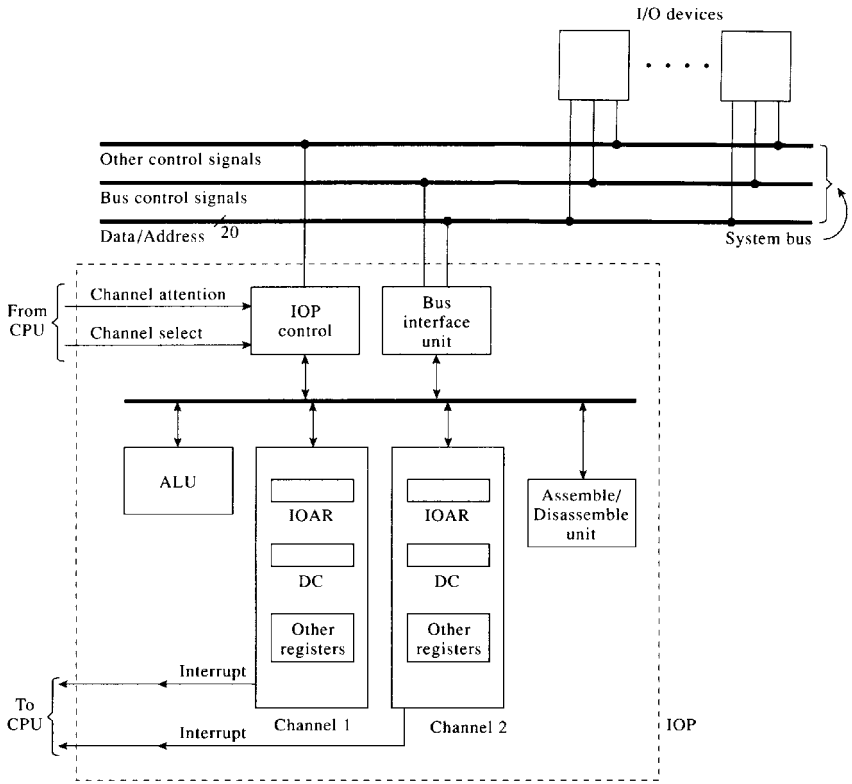



**Figure 6.39** CPU-8089 IOP interface

data/address bus onto which all the other peripheral devices and system memory are attached. The address and data are time-multiplexed on the 20-bit data/address bus. It is possible to configure a system with a separate I/O bus between the IOP and the peripheral devices as shown in Fig. 6.40, thereby reducing the traffic on the system bus. If such a local I/O bus is used, the IOP programs can be stored in the local memory attached to the local bus, thereby further reducing the traffic on the system bus.

Figure 6.41 shows the memory organization needed for the CPU/IOP communications. Corresponding to each DMA channel in the IOP, there is a channel control block (CB) consisting of a pointer to that channel's parameter block (PB), a channel status word, and a channel command word. The PB pointer points to the PB of the channel, which contains a pointer to the I/O program and all the required parameters for the I/O program. These parameters correspond to the device addresses, memory buffer addresses, and so on.

The CPU first tests the channel through the status word, establishes the command word and parameters in the PB, and commands the channel to start the I/O through the ATTENTION CHANNEL signal. The

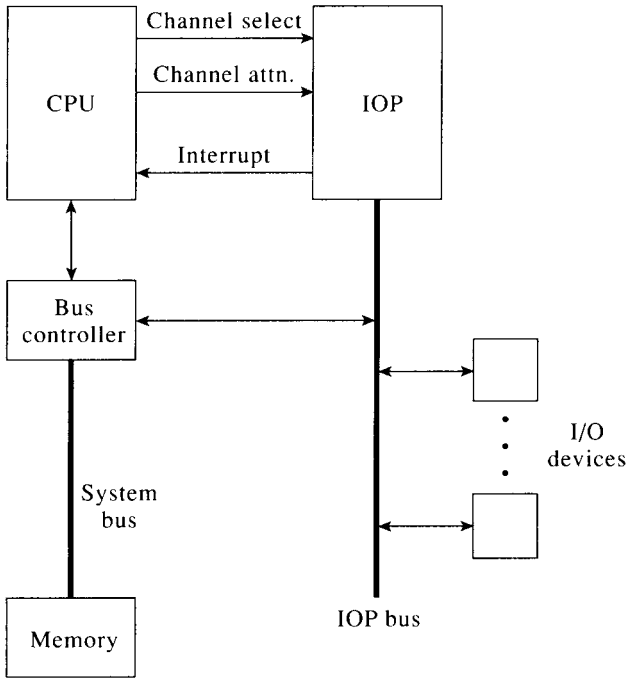



**Figure 6.40** CPU-IOP interface with separate I/O bus

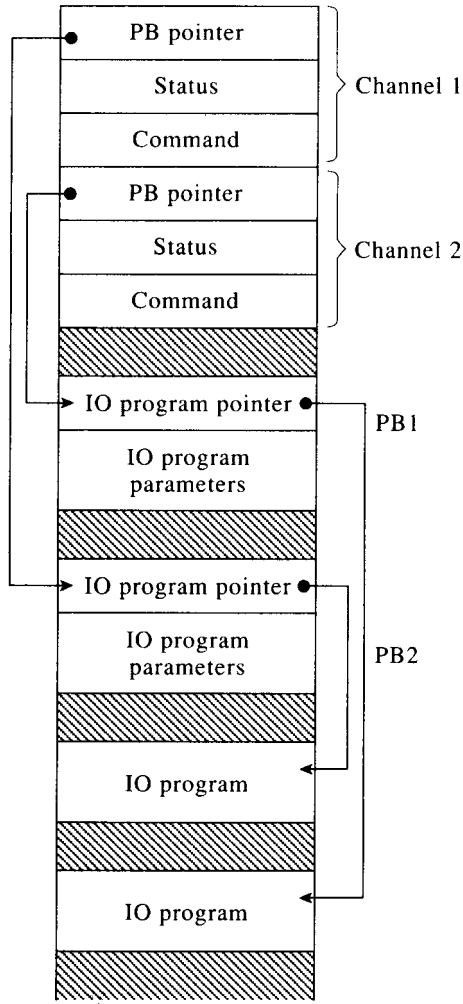



**Figure 6.41** Main memory organization for Intel 8089

CHANNEL SELECT is used to select one of the two channels. Once the I/O is complete, the channel produces an interrupt for the CPU.

The CPU uses its IN and OUT (if the IOP is configured in the I/O address space) or MOVE instructions (if the IOP is configured for memory mapped I/O) to establish command word and PB to sense the status of the IOP. The IOP instruction set consists of mainly data transfer instructions that move data between its internal registers and the system bus. It can

perform fixed-point arithmetic (add, increment, and decrement) and has a limited set of program control instructions. The IOP supports memory to memory block transfers in addition to the data transfer between I/O devices and the memory.

### 6.10.4 Intel 21285

This section is extracted from Intel Corporation's "21285 Core Logic for the SA-110 Microprocessor Datasheet" Sept 1998.

Figure 6.42 shows a block diagram of a system with an Intel 21285 I/O Processor connected to Intel SA-110 (Strong ARM®) microprocessor. SA-110 is optimized for embedded applications such as intelligent adapter cards, switches, routers, printers, scanners, RAID controllers, process control applications, set-top boxes, etc.

The Intel 21285 I/O Processor consists of the following components: synchronous dynamic random access memory (SDRAM) interface, read only memory (ROM) interface, peripheral component interconnect (PCI) interface, direct memory access (DMA) controllers, interrupt controllers, programmable timers, X-Bus interface, serial port, bus arbiter, joint test architecture group (JTAG) interface, dual address cycles (DAC) support, power management support.

The SDRAM controller controls from one to four arrays of synchronous DRAMs (SDRAMs) consisting of 8 Megabyte (MB), 16 MB, and 64 MB parts. All SDRAMs share command and address bits, but have separate clock and chip select bits as shown in Fig. 6.43.

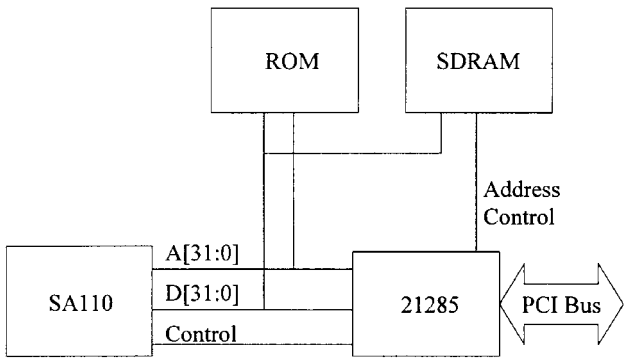



**Figure 6.42** System diagram

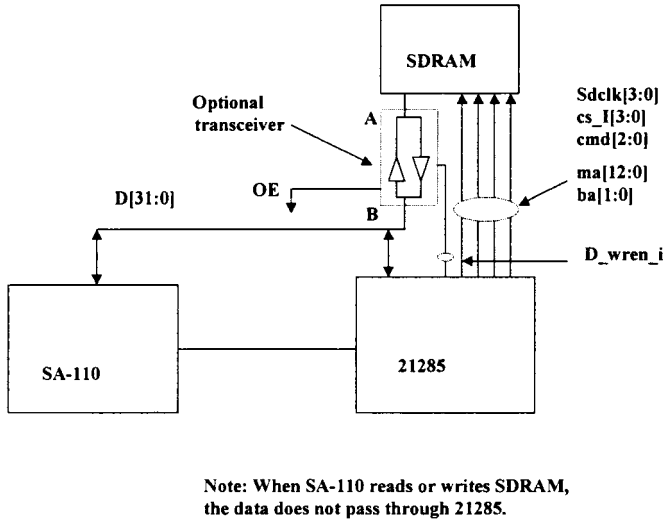



**Figure 6.43** SDRAM configuration

SDRAM operations performed by the 21285 are refresh, read, write, and mode register set. Reads and writes are generated by either the SA-110, PCI bus masters (including intelligent I/O ($I^2O$) accesses), or DMA channels. The SA-110 is stalled while the selected SDRAM bank is addressed. It is unstalled after the data has been latched into the SDRAM from D bus. The SA-110 is stalled while the selected SDRAM bank is addressed. It is unstalled when the first SDRAM data has been driven to the D bus. PCI memory write to SDRAM occurs if the PCI address matches the SDRAM base address register or the configuration space register (CSR) base address register, and the PCI command is either a memory write or a memory write and invalidate. PCI memory read from SDRAM occurs if the PCI address matches the SDRAM base address register or the CSR base address register, and the command is either a memory read, memory read line, or memory read multiple.

There are four registers controlling arrays 0 through 3. These four registers define each of the four SDRAM arrays' start address, size, and address multiplexing. Software must ensure that the arrays of SDRAM are mapped so there is no overlap of addresses. The arrays do not need to all be the same size; however, the start address of each array must be naturally aligned to the size of the array. The arrays do not need to form a contiguous address space, but to do so with different size arrays, place the largest array at the lowest address, next largest array above, and so on.

Figure 6.44 shows the ROM configuration. The ROM output enable and write enable are connected to address bits [30:31] respectively. The ROM address is connected to address bits [24:2].

The ROM can always be addressed by the SA-110 at 41000000h through 41FFFFFFh. After reset, the ROM is also aliased at every 16 megabytes throughout memory space, blocking access to SDRAM. This allows the SA-110 to boot from ROM at address 0. After any SA-110 write, the alias address range is disabled.

The SA-110 is stalled while the ROM is read. Each data word may require one, two, or four ROM reads, depending on the ROM width. The data is collected and packed into data words to be driven onto D[31:0]. When the ROM read complete, the 21285 unstalls the SA-110. The SA-110 is stalled while the ROM is written. The ROM write data must be placed on the proper byte lanes by software running on the SA-110, that is, the data is not aligned in hardware by the 21285. Only one write is done, regardless of the ROM width. When the ROM write completes, the 21285 unstalls the SA-110. PCI memory write to ROM occurs when the PCI address matches the expansion ROM base address register, bit [0] of the expansion ROM base address register is a 1, and the PCI command is either a memory write or a memory write and invalidate. The PCI memory write address and data is collected in the inbound first-in, first-out (FIFO) to be written to ROM at a later time. The 21285 target disconnects after one data phase. PCI memory read to ROM occurs when the PCI address matches the expansion ROM base address register, bit [0] of the expansion ROM base address register is a

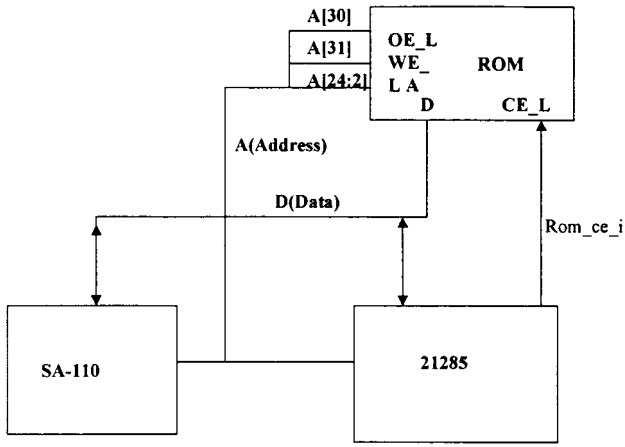



**Figure 6.44** ROM configuration

1, and the PCI command is either a memory read, memory read line, or memory read multiple. Timing during ROM accesses can be controlled by values in the SA-110 control register. The ROM access time, burst time, and tri-state time can be specified.

The 21285 is a programmable, two-way DMA channel which can move blocks of data from SDRAM to PCI or PCI to SDRAM. The DMA channels read parameters from a list of descriptors in memory, perform the data movement, and stop when the list is exhausted. For DMA operations, the SA-110 sets up the descriptors in SDRAM. Figure 6.45 shows DMA descriptors in local memory. Each descriptor occupies four data words and must be naturally aligned. The channels read the descriptors from local memory into working registers.

There are several registers for each channel: byte count register, PCI address register, SDRAM address register, descriptor pointer register, control register and DAC address register. The four data words provide the following information:

- The number of bytes to be transferred and the direction of transfer
- The PCI bus address of the transfer
- The SDRAM address of the transfer
- The address of the next descriptor in SDRAM or the DAC address.

The SA-110 writes the address of the first descriptor into the DMA channel *n* descriptor pointer register, writes the DMA channel *n* control register with other miscellaneous parameters, and sets the channel enable bit. If the channel initial descriptor in register bit [4] is clear, the channel reads the descriptor block into the channel control, channel PCI address,

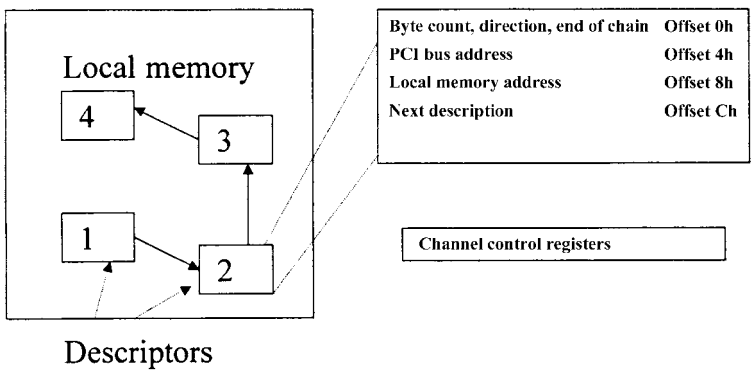



**Figure 6.45** Local memory descriptors

channel SDRAM address, and channel descriptor point registers. The channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit [2] in the DMA channel *n* control register. If the end of chain bit [31] in the DMA channel *n* byte count register (which is in bit [31] of the first word of the descriptor) is clear, the channel reads the next descriptor and transfers the data. If it is set, the channel sets the chain done bit [7] in the DMA channel *n* control register and then stops.

The PCI arbiter receives requests from five potential bus masters (four external and the 21285) and a grant is made to the device with the highest priority. The main register for the Bus Arbiter is X-Bus Cycle/Arbiter Register. Its offset is 148h and its function is used either to control the parallel port (X-Bus) or the internal PCI Arbiter.

There are two levels of priority groups with the low-priority groups as one entry in the high-priority group. Priority rotates evenly among the low-priority groups. Each device, including the 21285, can appear either in the low-priority group or the high-priority group, according to the value of the corresponding priority bit in the arbiter control register. The master is in the high-priority group if the bit is 1 and low-priority group if the bit is a 0. Priorities are reevaluated every time frame_1 is asserted, at the start of each new transaction on the PCI bus. The arbiter grants the bus to the higher priority device in the next clock style if it interrupts when a lower-priority device is using the bus. The master that initiated the last transaction has the lowest priority in the group.

PCI is a local bus standard (a data bus that connects directly to the microprocessor) developed by Intel Corporation. Most modern PCs include a PCI bus in addition to a more general Industry Standard Architecture (ISA) expansion bus. Many analysts believe that PCI will eventually supplant ISA entirely. PCI is a 64-bit bus, though it is usually implemented as a 32-bit bus. It can run at clock speeds of 33 or 66M hertz (Hz). At 32 bits and 33 MHz, it yields a throughput rate of 133M bits per second (bps). PCI is not tied to any particular family of microprocessors. It acts like a tiny "local area network" inside the computer, in which multiple devices can each talk to each other, sharing a communication channel that is managed by the chipset. The PCI target transactions include the following:

- Memory write to SDRAM
- Memory read, memory read line, memory read multiple to SDRAM
- Type 0 configuration write
- Type 0 configuration read
- Write to CSR
- Read to CSR
- Write to $I^2O$ address

- Read to $I^2O$ address
- Memory read to ROM.

The PCI master transactions include the following:

- Dual address cycles (DAC) support from SA-110 or DMA
- Memory write, memory write and invalidate from SA-110 or DMA
- Selecting PCI command for writes
- Memory write, memory write and invalidate from SA-110 or DMA
- Selecting PCI command for writes
- Memory read, memory read line, memory read multiple from SA-110 or DMA
- I/O write
- I/O read
- Configuration write
- Special cycle
- Interrupt acknowledge (IAC) read.
- PCI request operation
- Master latency timer.

The message unit provides a standardized message-passing mechanism between a host and a local processor. It provides a way for the host to read and write lists over the PCI bus at offsets of 40h and 44h from the first base address. The function of the FIFOs is to hold message frame addresses, which are offsets to the message frames. The $I^2O$ message units supports four logical FIFOs:

| Name | Function |
|---|---|
| Inbound free_list<br>Inbound post_list | Manage I/O requests from the host processor to SA-110 |
| Outbound free_list<br>Outbound post_list | Manages messages that are replies from SA-110 to the host processor |

The $I^2O$ inbound FIFOs are initiated by the SA-110. The host sends messages to the local processor. The local processor operates on the messages. The SA-110 allocates memory space for both the inbound free_list and post_list FIFOs and initializes the inbound pointers. The SA-110 initializes the inbound free_list by writing valid Memory Frame Address (MFA) values to all entries and increment the number of entries in the inbound free_list count register. The $I^2O$ outbound FIFOs are initialized by the SA-

110. The SA-110 allocates memory space for both outbound free_list and outbound post_list FIFOs. The host processor initializes the free_list FIFO by writing valid MFAs to all entries. See Figs. 6.46 and 6.47.

The 21285 contains four 24-bit timers. There are control and status registers for each timer. A timer has two modes of operation: free-run and periodic. The timers and their interrupt can be individually enabled/disabled. The interrupts can be enabled/disabled by setting the appropriate bit(s) in the interrupt request (IRQ) enable set/fast interrupt request (FIQ) enable set registers. Timer 4 an be used as a watchdog timer, but requires that the watchdog enable bit be set in the SA-110 control register. Figure 6.48 shows a block diagram of a typical timer. The control, status and data registers for this capability are accessible only from the SA-110 interface (all offsets are from 4200 0000H).

There are three registers that govern the operations of a timer. In the periodic mode, the Load register contains a value that is loaded into a down counter and decremented to zero. The Load register is not used in the free-run mode. The control register allows the user to select a clock rate that will be used to decrement the counter, the mode of operation and whether the timer is enabled. The clear register resets the timer interrupt.

The 21285 has only one register that might be considered a status register and that is the value register. It can be read to obtain the 24-bit value in the down counter.

In terms of the software interface the user needs to determine:

- The clock speed that will be used to decrement the down counter
- Whether free-run or periodic mode will be used
- Whether to generate an IRQ or FIQ interrupt or both.

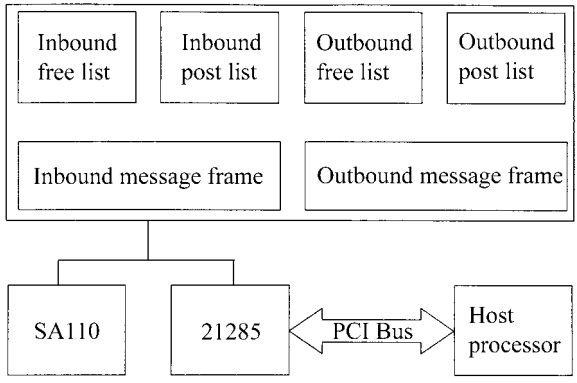



**Figure 6.46** $I^2O$ message units

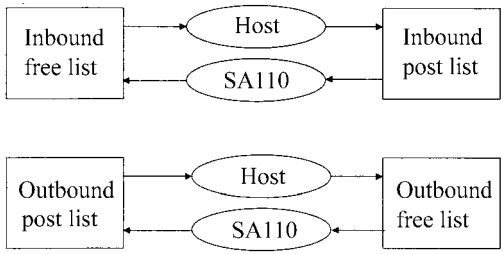



**Figure 6.47** Configuration of MFAs

Once determined, the clock speed, mode and enable bit for the timer must be written into the timer's control register. The interrupt type must be enabled by setting the appropriate bit(s) in the IRQ enable set/FIQ enable set registers.

In the free-run mode, a down counter is loaded with the maximum 24-bit value and decremented until it reaches zero (0), at which time an interrupt is generated. The periodic mode loads the down counter with the value in the load register and decrements until it reaches zero (0), at which time an interrupt is generated. Upon reaching zero (0), timers in the free-run mode are reloaded with the maximum 24-bit value and periodic mode timers are reloaded from the load register. In both cases, once reloaded, the counters decrement to zero (0) to repeat the cycle.

The 21285 has a universal asynchronous receiver transmitter (UART) that can support bit rates from approximately 225 bits per second (bps) to approximately 200K bps. The UART contains separate transmit and receive FIFOs which can hold 16 data entries each. The UART generates receive

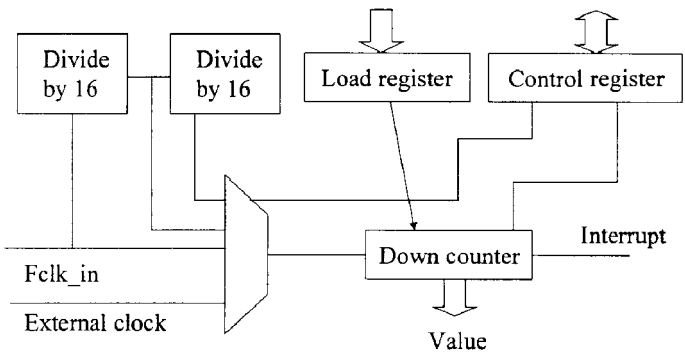



**Figure 6.48** Timer block diagram

and transmit interrupts that can be enabled/disabled by setting the appropriate bit(s) in the IRQ enable set/FIQ enable set registers. The control, status and data registers for this capability are accessible only from the SA-110 interface (all offsets are from 4200 0000H).

There are four registers that govern the operations of the UART. The H_UBRLCR register allows the user to set the data length, enable parity, select odd or even parity, select the number of stop bits, enable/disable the FIFOs and generate a break signal (the tx pin is held low). The M_UBRLCR and L_UBRLCR registers, together, contain the 12 bit, baud rate divisor (BRD) value. The UARTCON register contains the UART enable bit along with bits for using the SIREN HP SIR protocol and the infrared data (IrDa) encoding method.

The UART contains a single data register. It allows the transmit and receive FIFOs to be accessed. Data to/from the UART can be accessed either via FIFOs or as a single data word. The access method is determined by the enable FIFO bit in the H_UBRLCR control register.

The 21285 has two status registers: RXSTAT indicates any framing, parity or overrun errors associated with the received data. A read of RXSTAT must follow a read of UARTDR because RXSTAT provides the status associated with the last piece of data read from UARTDR. This order cannot be reversed. UARTFLG contains transmit and receive FIFO status and an indication whether the transmitter is busy.

When the UART receives a frame of data (start bit, data bits, stop bits and parity), the data bits are stripped from the frame and put in the receive FIFO. When the FIFO is more than half full, a receive interrupt is generated. If the FIFO is full, an overrun error will be generated. The framing data is examined and if incorrect a framing error is generated. Parity is checked and the parity error bit set accordingly. The data words are accessed by reading UARTDR and any errors associated can be read from RXSTAT.

When the UART transmits data, a data word is taken from the transmit FIFO and framing data is added (start bit, stop bits and parity). The frame of data is loaded into a shift register and clocked out. When the FIFO is more than half-empty, a transmit interrupt is generated.

The 21285 is compliant with the Institute of Electrical and Electronics Engineers (IEEE) 1149.1 "IEEE Standard Test Access Port and Boundary-Scan Architecture." It provides five pins to allow external hardware and software to control the test access port (TAP).

The 21285 provies an nIRQ and nFIQ interrupt to the SA110 processor. These interrupts are the logical NOR of the bits in the IRQ status and FIQ status, respectively. These registers collect the interrupt status for 27 interrupt sources. All the interrupts have the equal priority. The control,

status and data registers for this capability are accessible only from the SA-110 interface. Figure 6.49 is a block diagram of a typical interrupt.

The following control registers deal with interrupts: The IRQ enable set register allows individual interrupts to be enabled. The IRQ enable clear register disables an individual interrupt. The IRQ soft register allows the software to generate an interrupt. The following status registers to deal with interrupts: The IRQ enable register indicates which interrupts are being used by the system. The IRQ raw status register indicates which interrupts are active. The IRQ status register is the logical AND of the IRQ enable register and the IRQ raw status register.

When an interrupting device activates its interrupt, the information is loaded into the IRQ raw status register. This data is ANDed with the IRQ enable register to set bits in the IRQ status register. The bits in the IRQ status register are NORed together to generate the nIRQ interrupt to the SA110. The SA110 determines what caused the interrupt by reading the IRQ status register. An interrupt is cleared by resetting the appropriate bit in the IRQ enable clear register.

The 21285 provides an 8, 16 and 32 bit parallel interfaces with the SA110. The duration and timing relationship between the address, data and control signals is accomplished via control register settings. This provides the 21285 a great deal of flexibility to accommodate a wide variety of input/output devices. The control, status and data registers for this capability are accessible only from the SA-110 interface (all offsets are from 4200 0000H). Figure 6.50 shows a block diagram of the X-Bus interface.

The 21285 provides the following control and status registers to deal with the X-bus: The X-bus cycle register allows the user to: set the length of

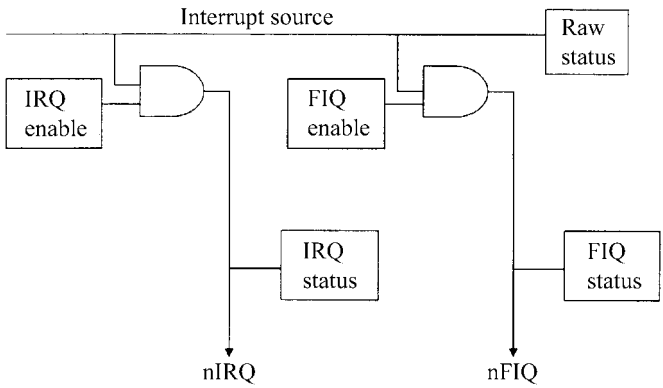



**Figure 6.49** Interrupt block diagram

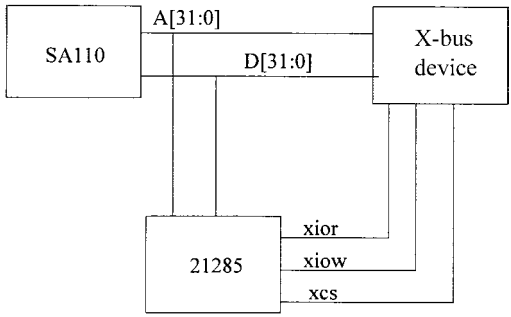



**Figure 6.50** Bus configuration block diagram

a read/write cycle; apply a divisor to the clock used during read and write cycles; choose a chip select; determine if the X-bus or PCI bus is being used; the X-bus and PCI interrupt levels. The X-bus I/O strobe register allows the user to control when the xior and xiow signals go low and how long they stay low within the programmed read/write cycle length.

### 6.10.5 DEC PDP-11 System

The PDP-11 system (Fig. 6.51) structure is based on the 56-line *unibus*. This bus connects the I/O devices, CPU, and memory. The devices can send, receive, or exchange data without processor intervention and without immediate data buffering in the memory. The unibus makes the peripheral interfacing requirements uniform and hence made the PDP-11 family of computers popular. Several unibus-compatible peripheral controllers and interfaces are available from non-DEC sources.

PDP-11 architecture takes advantage of the unibus in its method of addressing peripheral devices. Memory elements, such as the main core memory or any read-only or solid-state memories, have ascending addresses starting at zero, while registers that store I/O data or the status of individual peripheral devices have addresses in the highest 4K words of addressing space. Two memory words are normally reserved for each peripheral device (one for data, one for control). Some devices, such as magnetic tape and disk units, use up to six words.

Communication between any two devices on the bus is performed in a master–slave mode. During any bus operation, one device, the bus master, controls the bus when communicating with another device on the bus, the slave. For example, the processor as master can fetch an instruction from the memory, which is always a slave; or the disk as master can

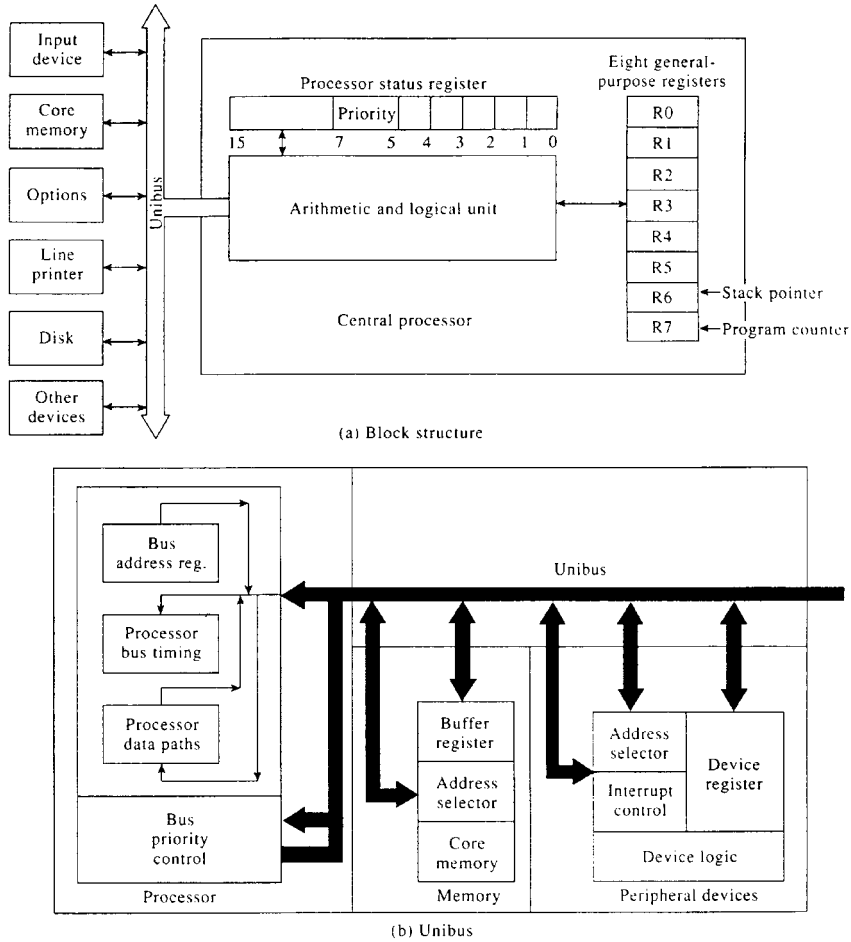



**Figure 6.51** PDP-11 system (Copyright, Digital Equipment Corporation, 1978. All Rights Reserved.)

transfer data to the memory as slave. Master–slave relationships are dynamic; that is, the processor may, for example, pass bus control to a disk at which point the disk may become master and communicate with slave memory, and so on.

When two or more devices try to obtain control of the bus at one time, priority circuits decide among them. Devices have unique priority levels, fixed at system installation. A unit with a high priority level always takes precedence over one with a low priority level; in the case of units with equal

priority levels, the one closest to the processor on the bus takes precedence over those further away (daisy chain).

Suppose the processor has control of the bus when three devices, all of higher priority than the processor, request bus control. If the requesting devices are of different priority, the processor will grant use of the bus to the one with the highest priority. If they are all of the same priority, all three signals come to the processor along the same bus line and the processor sees only one request signal. Its reply, granting priority, travels down the bus to the nearest requesting device, passing through any intervening nonrequesting device. The requesting device takes control of the bus, executes a single bus cycle of a few hundred nanoseconds, and relinquishes the bus. Then the request grant sequence occurs again, this time going to the second device down the line, which has been in wait mode. When all higher priority requests have been granted, control of the bus returns to the lowest priority device.

The processor usually has the lowest priority because in general it can stop whatever it is doing without serious consequences. Peripheral devices may be involved with some kind of mechanical motion or may be connected to a real-time process, either of which requires immediate attention to a request to avoid data loss.

The priority arbitration takes place asynchronously in parallel with data transfer. Every device on the bus except memory is capable of becoming a bus master. Communication is interlocked, so that each control signal issued by the master must be acknowledged by a response from the slave to complete the transfer. This simplifies the device interface because timing is no longer critical. The maximum typical transfer rate on the unibus is one 16-bit word ever 400 ns, or about 2.5 million 16-bit words per second.

### Bus Control

There are two ways of requesting bus control: nonprocessor request (NPR) or bus request (BR). An NPR is issued when a device needs to perform a data transaction with another device and does not use the CPU. Therefore, the CPU can relinquish bus control while an instruction is being executed. A BR is issued when a device needs to interrupt the CPU for service and hence an interrupt is not serviced until the processor has finished executing its current instruction. The following protocol is used to request the bus (BR):

1. The device makes a bus request by asserting the BR line.
2. Bus arbitrator recognizes the request by issuing a bus grant (BG). This bus grant is issued only if the priority of the device is greater than the priority currently assigned to the processor.

3. The device acknowledges the bus grant and inhibits further grants by asserting SELECTION ACKNOWLEDGE (SACK). The device also clears BR.
4. Bus arbitrator receives SACK and clears BG.
5. The device asserts BUS BUSY (BBSY) and clears SACK.
6. The device asserts BUS INTERRUPT (INTR) and sends its address vector.
7. The processor enters an interrupt service routine corresponding to the device, based on the address vector.

The following protocol is used for NPR data transfer requests:

1. The device makes a nonprocessor request by asserting NPR.
2. Bus arbitrator recognizes the request by issuing a nonprocessor grant or NPG.
3. The device acknowledges the grant and inhibits further grants by asserting SACK; device also clears NPR.
4. Bus arbitrator receives SACK and clears NPG.
5. The device asserts BUS BUSY (BBSY) and clears SACK. (Once a device's bus request has been honored, it becomes bus master as soon as the current bus master relinquishes control.)
6. Current bus master relinquishes bus control by clearing BBSY.
7. New device assumes bus control by setting BBSY and starts its data transfer to the slave device.

### Interrupt Structure

Figure 6.52 shows the interrupt structure of PDP-11. There are five levels of priority: NPR, BR7, BR6, BR5, and BR4. NPR has the highest priority and BR4 has the lowest. In addition, there are interrupt levels 0 through 3 for software interrupts. The CPU itself can operate at any of the priority levels 0 through 7. The CPU releases the bus to a requesting device if the device priority is higher than the CPU priority at that time. There can be several devices connected to same priority level. Within each level, the devices are daisy chained.

There are two lines associated with each level: the bus request BR (BR7 through BR4) and the bus grant BG (BG7 through BG4). Because there are only five vertical priority levels, it is often necessary to connect more than one device to a single level.

The grant line for the NPR level (NPG) is connected to all the devices on that level in a daisy chain arrangement. When an NPG is issued, it first goes to the device closest to the CPU. If that device did not make the

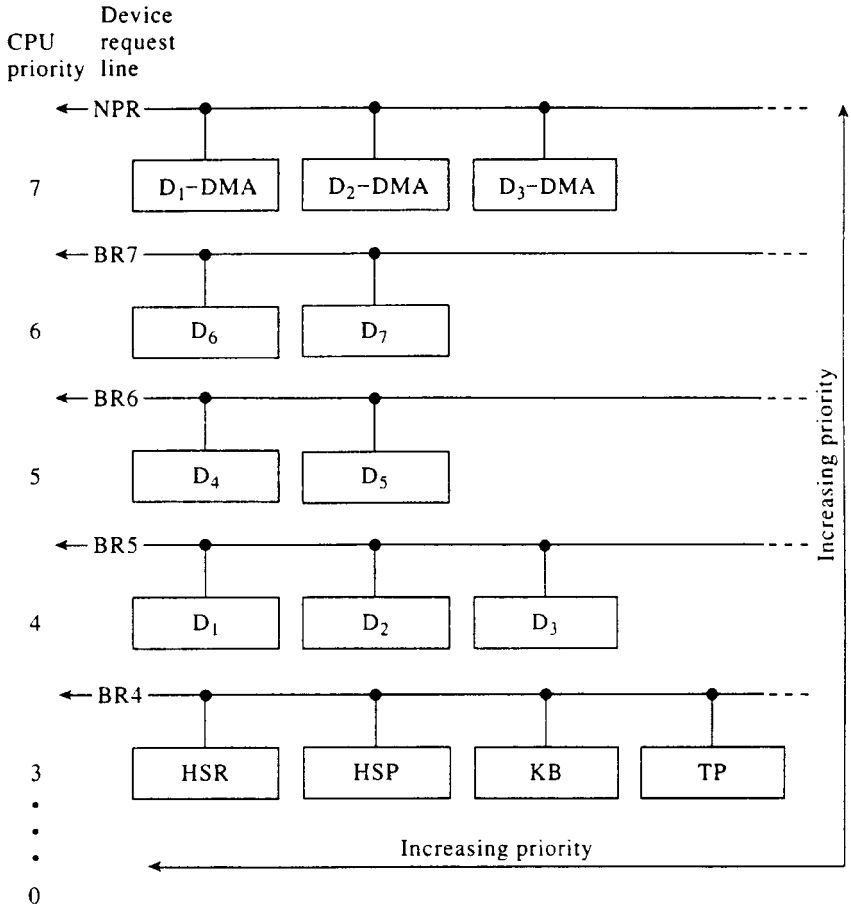



**Figure 6.52** Priority control in PDP-11 (Copyright, Digital Equipment Corporation, 1978. All Rights Reserved.)

request, it permits the NPG to travel to the next device. Whenever the NPG reaches a device that has made a request, that device captures the grant and prevents it from passing to any subsequent device in the chain. BR chaining is identical to NPR chaining in function. However, each BR level has its own BG chain.

The CPU can be set to any one of eight priority levels. Priority is not fixed; it can be raised or lowered by software. The CPU priority is elevated from level 4 to level 6 when the CPU stops servicing a BR4 device and starts servicing a BR6 device. This programmable priority feature permits masking

of bus requests. The CPU can hold off servicing lower priority devices until more critical functions are completed.

### Interrupt Service

The interrupting device first gains the bus control and sends its vector to the CPU. The vector points to the two memory locations reserved for the device. The first of these memory words contains the address of the interrupt service routine (the new program counter value) and the second contains the new program status word (PS). The CPU pushes the current PC and PS contents onto a stack and loads the new PC and PS values, to enter the interrupt service routine. At the end of service, the return from interrupt instruction pops two words (PC and PS) into the corresponding CPU registers.

## 6.10.6 Control Data 6600

The CDC 6600 first announced in 1964 was designed for two types of use: large-scale scientific processing and time-sharing of smaller problems. To accommodate large-scale scientific processing, a high-speed, floating-point, multifunctional CPU was used. The peripheral activity was separated from CPU activity by providing twelve input/output channels controlled by ten peripheral processors. Figure 6.53 shows the system structure.

The ten peripheral processors have access to central memory. One of these processors acts as a control processor for the system while the others are performing I/O tasks. Each of these processors has its own memory, used for storing programs and for data buffering. The peripheral processors access the central memory in a time-shared manner through the barrel mechanism. The barrel mechanism has a 1000 ns cycle. Each peripheral processor is connected to the central memory for 100 ns in each cycle of the barrel. The peripheral processor provides the memory-accessing information to the barrel during its 100 ns time slot. Actual memory access takes place during the remaining 900 ns of the barrel cycle, and the peripheral processor can start a new access during its next time slot. Thus, the barrel mechanism handles 10 peripheral processor requests in an overlapped manner. I/O channels are 12-bit bidirectional paths. Thus, one 12-bit word can be transferred into or out of the memory every 1000 ns by each channel.

We will discuss the central processor structure of CDC 6600 further in Chapter 10.

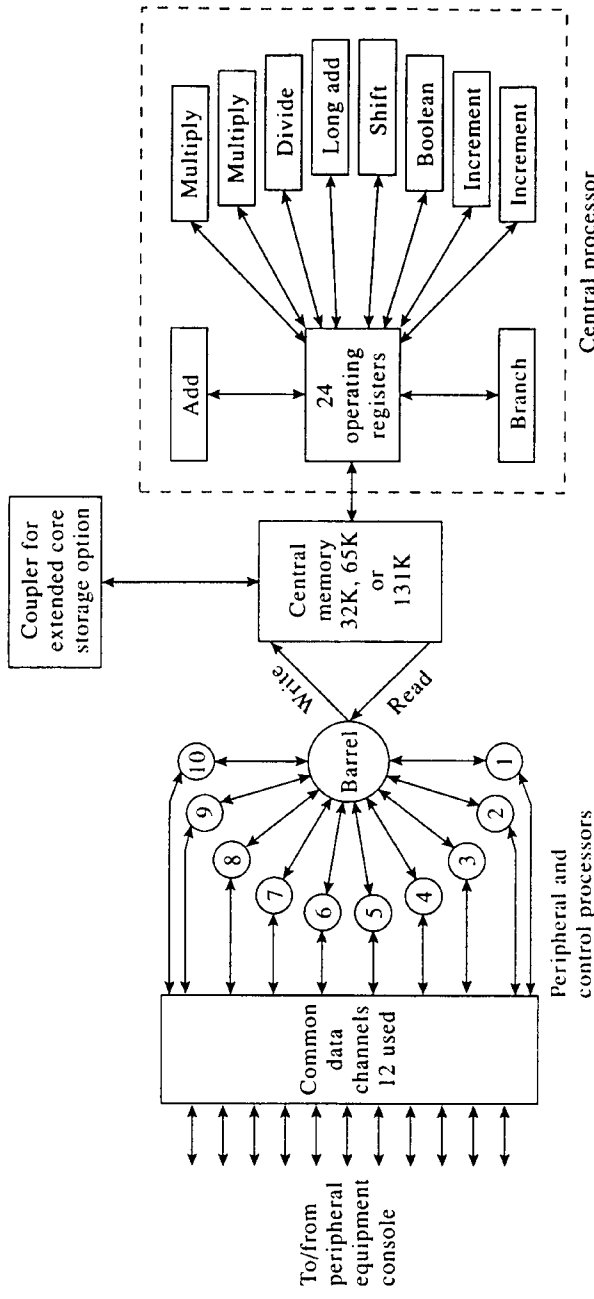



**Figure 6.53** CDC 6600 system structure (Courtesy of Control Data Corporation)

## 6.11 SUMMARY

Various modes of data transfer between I/O devices and the CPU were discussed in this chapter. The advances in hardware technology and dropping prices of hardware have made it possible to implement cost-effective and versatile I/O structures. Details of the I/O structures of representative commercially available machines were provided. Interfacing I/O devices to the CPU is generally considered a major task. However, the recent efforts in standardizing I/O transfer protocols and the emergence of standard buses have reduced the tedium of this task. It is now possible to easily interface an I/O device or a CPU made by one manufacturer with compatible devices from another manufacturer. The trend in delegating the I/O tasks to I/O processors has continued. In fact, modern computer structures typically contain more than one general-purpose processor, some of which can be dedicated to I/O tasks, as the need arises. This chapter has provided details of representative I/O devices. Newer and more versatile devices are being introduced daily. Any list of speed and performance characteristics of these devices provided in a book would quickly become obsolete. Refer to the magazines listed in the References section for up-to-date details.

Several techniques are used in practical architectures to enhance the I/O system performance. Each system is unique in its architectural features. The concepts introduced in this chapter form the baseline features that help to evaluate practical architectures. In practice, the system structure of a family of machines evolves over several years. Tracing the family of machines to gather the changing architectural characteristics and to distinguish the influence of advances in hardware and software technologies on the system structure would be an interesting and instructive project.

## REFERENCES

*21285 Core Logic for the SA-110 Microprocessor Data Sheet*, Santa Clara, CA: Intel, 1998.

Baron, R. J. and Higbie, L. *Computer Architecture*, Reading, MA: Addison Wesley, 1992.

*Computer*, Los Alamitos, CA: IEEE Computer Society, Published monthly.

*Computer Design*, Northbrook, IL: PennWell Publishing, Published monthly.

Cramer, W. and Kane, G., *68000 Microprocessor Handbook*, Berkeley, CA: Osborne McGraw-Hill, 1986.

DiGiacomo, J. *Digital Bus Handbook*, New York, NY: McGraw-Hill, 1990.

*EDN*, Highlands ranch, CO: Cahners, Published twice monthly.

*Electronic Design*, Hasbrouck Heights, NJ: Penton Publishing, Published twice monthly.
Hill, M. D., Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.
*IEE/ANSI 796 Standard*, New York, NY: IEEE, 1983.
*IEE/ANSI 1296 Standard*, New York, NY: IEEE, 1988.
*MC68000 Users Manual*, Austin, TX: Motorola, 1982.
*MCS-80 Users Manual*, Santa Clara, CA: Intel, 1977.
Mano, M. M. *Computer Systems Architecture*, Englewood Cliffs, NJ: Prentice-Hall, 1992.
Patterson, D. A. and Hennessey, J. L. *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann, 1990.
Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.
*Simple Multibus II I/O Replier*, Mountainview, CA: PLX Technology, 1989.
Stone, H. S. *High Performance Computer Architecture*, Reading, MA: Addison Wesley, 1993.
Tannenbaum, A. S. and Goodman, J. R. *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall, 1998.

## PROBLEMS

6.1 What is the maximum number of I/O devices that can be interfaced to ASC, given its current instruction format?
6.2 It is required to interface 5 input devices (device address 0 through 4) and 7 output devices (device address 5 through 11) to ASC. Show the complete hardware needed.
6.3 A computer system has a 16-bit address bus. The first 4K of the memory must be configured with ROM and the remaining as RAM. Show the circuit that will generate an error if an attempt is made to write into the ROM area.
6.4 Design a priority encoder with 8 inputs. The 3-bit output of the encoder indicates the number of the highest priority input that is active. Assume that input 0 has the lowest priority and 7 has the highest priority.
6.5 What is the maximum number of I/O ports that can be implemented using MC68230s on a system bus with 16 address lines?
6.6 Design an 8-bit input port and an output port using MC68230 for the MC68000. The input port receives a 4-bit BCD value on its low-order bits, and the output port drives a 7-segment display from its bits. Show the assembly language program needed.
6.7 Rewrite the fetch microinstruction sequence for ASC, assuming that the processor status is saved on a stack before entering interrupt service. Include a stack pointer register.
6.8 Rewrite the sequence in Problem 6.7, assuming a vectored interrupt mode. Assume that the device sends its address as the vector.

6.9 Write microprograms to describe the handshake between the peripheral and CPU during (a) interrupt mode I/O and (b) DMA. Use the HDL structure common to ASC microprograms to describe these asynchronous operations.

6.10 Develop a complete hardware block diagram of a multiplexer channel. List the microoperations of each block in the diagram.

6.11 Repeat Problem 6.10 for a selector channel.

6.12 Develop a schematic for a printer controller. Assume that one character (8 bits) is transferred to the printer at a time, in a programmed I/O mode.

6.13 Repeat Problem 6.12, assuming that the printer has a buffer that can hold 80 characters. Printing is activated only when the buffer is full.

6.14 Design a bus controller to resolve the memory-access requests by the CPU and a DMA device.

6.15 The structure of Fig. 6.12 polls the peripheral devices in the order of the device numbers. Design a structure in which the order of polling (i.e., priorities) can be specified. (Hint: Use a buffer to store the priorities.)

6.16 Design a set of instructions to program the DMA controller. Specify the instruction formats, assuming ASC as the CPU.

6.17 A cassette recorder-player needs to be interfaced to ASC. Assume that the cassette unit has a buffer to store 1 character and the cassette logic can read or record a character into or out of this buffer. What other control lines are needed for the interface? Generate the interface protocol.

6.18 Compare the timing characteristics of programmed input mode with that of the vector-mode, interrupt-driven input scheme, on ASC. Use the major cycles needed to perform the input as the unit of time.

6.19 Assume that there is a serial output line connected to the LSB of the accumulator. The contents of the accumulator are to be output on this line. It is required that a start bit of 0 and 2 stop bits (1) delimit the data bits. Write an assembly language program on ASC to accomplish the data output. What changes in hardware are needed to perform this transfer?

6.20 Look up the details of the power-failure detection circuitry of a computer system you have access to, to determine how a power-failure interrupt is generated and handled.

6.21 There are memory-port controllers that allow interfacing multiple memories to computer systems. What is the difference between a memory-port controller and a bus controller?

6.22 Suppose a processor receives multiple interrupt requests at about the same time, it should service all of them before returning control to the interrupted program. What mechanism is ideal for holding these multiple requests: stack, queue or something else? Why?

6.23 A processor executes 1000 K instructions per second. The bus system allows a bandwidth of five megabytes per second. Assume that each instruction requires on average eight bytes of information for the instruction itself and the operands. What bandwidth is available for the DMA controller?

6.24 If the processor of Problem 6.23 performs programmed I/O and each byte of I/O requires two instructions, what bandwidth is available for the I/O?

6.25 In this chapter, we assumed that the interrupt service always starts at the beginning of the instruction cycle. It may not be desirable to wait until the end of current instruction cycle during complex instructions (such as multiple move, etc.). What are the implications of interrupting an instruction cycle in progress? How would you restart the interrupted instruction?

# 7
# Processor and System Structures

The main objective of Chapter 5 was to illustrate the design of a simple but complete computer (ASC). Simplicity was our main consideration, and so architectural alternatives possible at each stage in the design were not presented. Chapter 6 extended the I/O subsystem of ASC in light of I/O structures found in practical machines. This chapter describes selected architectural features of some popular machines as enhancements to ASC architecture. We will concentrate on the details of instruction set, addressing modes, and register and processor structures. Architectural enhancements to the memory, control unit, and ALU are presented in subsequent chapters. For the sake of completeness, we will also provide in this chapter a brief description of a popular series of microprocessors (Intel Pentium) and a microcomputer system based on these microprocessors. We will first distinguish between the four popular types of computer systems now available.

## 7.1 TYPES OF COMPUTER SYSTEMS

A modern-day computer system can be classified as a *supercomputer*, a *large-scale machine* (or a *mainframe*), a *minicomputer*, or a *microcomputer*. Other combinations of these four major categories (mini–micro system, micro–mini system, etc.) have also been used in practice. It is difficult to produce sets of characteristics that would definitively place a system in one of these categories today; original distinctions are becoming blurred with advances in hardware technology. A microcomputer of the 1980s, for example, can provide roughly the same processing capability as that of a large-scale computer system of the 1970s. Nevertheless, Table 7.1 lists some of the characteristics of the four classes of computer systems. As can be noted, there is a considerable amount of overlap in their characteristics. A supercomputer is defined as the most powerful computer system available to date.

**Table 7.1** Characteristics of Contemporary Computers

| Characteristic | Supercomputer | Large-scale computer | Minicomputer | Microcomputer |
|---|---|---|---|---|
| CPU size | Not easily portable | Not easily portable | 2 ft × 2ft (60 cm × 60 cm) rack | Small circuit card (or one IC) |
| Relative cost of minimum system | 1000s | 100s | 10s | 1 |
| Typical Word size (bits) | 64 | 32–64 | 16–32 | 4–32 |
| Typical processor cycle time ($\mu$s) | 0.01 | 0.5–0.75 | 0.2–1 | 1 |
| Typical application | High-volume scientific | General purpose | Dedicated or general purpose | Dedicated or general purpose |

The physical size of a "minimum configuration" system is probably still a distinguishing feature: A supercomputer or other large-scale system would require a wheelbarrow to move it from one place to another, while a minicomputer can be carried without any mechanical help, a microcomputer can fit easily on an 8- by 12-inch board (or even on one IC) and can be carried in one hand.

With the advances in hardware and software technology, architectural features found in supercomputers and large-scale machines eventually appear in mini- and microcomputer systems; hence, the features discussed in this and subsequent chapters of this book are assumed to apply equally well to all classes of computer systems.

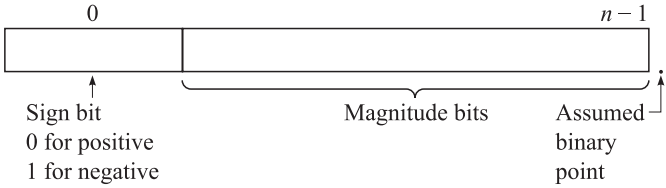
## 7.2 OPERAND (DATA) TYPES AND FORMATS

The selection of processor word size is influenced by the types of data (or operands) that are expected in the application for the processor. The instruction set is also influenced by the various types of operands allowed. Data representation differs from machine to machine. The data format depends on the word size, code used (ASCII or EBCDIC), and the arithmetic (1s or 2s complement) mode employed by the machine. The most common operand or data types are: fixed-point (or integer) binary, decimal (or BCD),

floating-point (or real) binary, and character strings. Typical representation formats for each of these types are described below.

### 7.2.1 Fixed Point

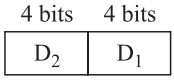
The fixed-point binary number representation is common to all the machines and is shown below for an $n$-bit machine:



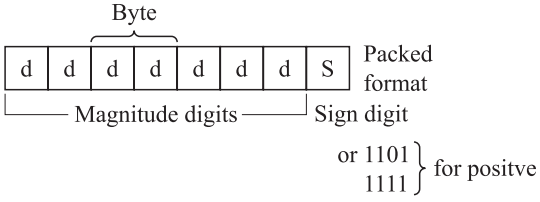

Negative numbers are represented either in 2s complement or in 1s complement form, the 2s complement form being the most common.

### 7.2.2 Decimal Data

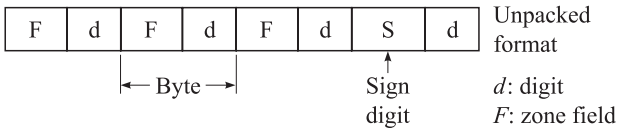
Machines that allow decimal (BCD) arithmetic mode use 4 -bits per decimal digit and pack as many digits as possible into the machine word. In some machines, a separate set of instructions is used to operate on such data. In others, the arithmetic mode is changed from BCD to binary (and vice versa) by an instruction provided for that purpose. Thus, the same arithmetic instructions can be used to operate on either type of data. A BCD digit occupies 4 bits and hence two decimal (BCD) digits can be packed into 1 byte as shown below:




IBM 370 allows the decimal numbers to be of variable length (from 1 to 16 digits). The length is specified (or implied) as part of the instruction. Two digits are packed into one byte. Zeros are padded on the most significant end if needed. A typical data format is shown below:



In the unpacked form, IBM 370 uses one byte for each decimal digit: the upper 4 bits is the zone field and contains 1111 and the lower 4 bits have the decimal digit.



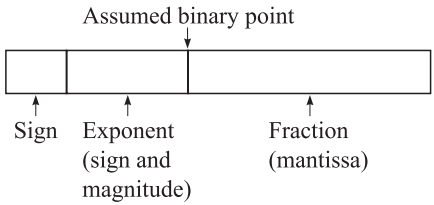

All the arithmetic operations are on packed numbers; the unpacked format is useful for input/output since the IBM 370 uses EBCDIC code (one byte per character).

### 7.2.3 Character Strings

These are represented one byte per character, using either ASCII or EBCDIC code. The maximum length of the character string is a design parameter of the particular machine.

### 7.2.4 Floating-point Numbers

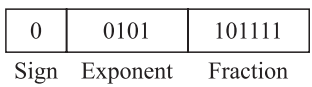
We have so far used only integer arithmetic in this book. In practice, floating-point (real) numbers are used in computations and are represented as shown below:




The representation consists of the sign of the number, an exponent field, and a fraction field. For example, consider the real number $(23.5)_{10}$. This number can be represented as:

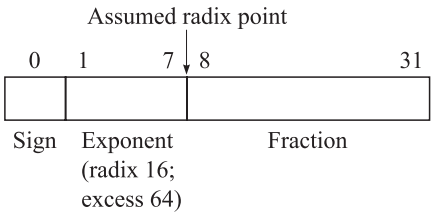
$$\begin{aligned}(23.5)_{10} &= (10111.1)_2 \\ &= 0.101111 \times 2^5\end{aligned}$$

and hence the representation is



In the above example, the most significant bit of the fraction is 1. When the fraction consists of 1 or more zero bits at the most significant end, it is usually shifted left until the most significant bit is non-zero and the exponent is adjusted accordingly. This process is called *normalization* of a fraction. Since in practice there will be only a limited number of bits to represent the fraction, normalization enables more bits to be retained, thus increasing accuracy. Two floating-point representations are described below.

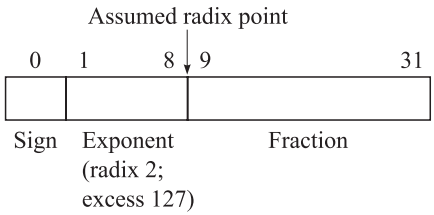
## IBM 370




The fraction is represented using either 24 bits (single precision) or 56 bits (double precision). The exponent is a radix-16 exponent and is expressed as an excess-64 number (that is, a 64 is added to the true exponent so that the exponent representation ranges from 0 through 127 rather than −64 to 63).

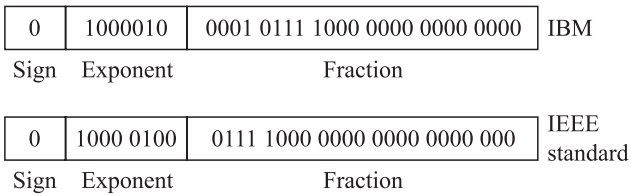
## IEEE Standard

The Institute of Electrical and Electronic Engineers' (IEEE) standard for floating-point representation is shown below:



**Example 7.1** The number $(23.5)_{10}$ is shown below in both of the representations:

$$\begin{aligned}(23.5)_{10} &= (10111.1)_2 = (0.101111) \times (2)^5 \\ &= (0001\ 0111.1)_2 \\ &= 0.(0001\ 0111\ 1)_2 \times (16)^2\end{aligned}$$




The first bit indicates the sign of the number. The 8-bit exponent field represents a base-2 exponent in excess-127 format. Remaining 23 bits are used to represent the fraction. Since the fraction is always normalized, the most significant bit is 1. This bit is assumed and not specifically represented. As such, the 23-bit fraction field actually represents a 24-bit fraction.

## 7.3 INSTRUCTION SET

The selection of a set of instructions for a machine is influenced heavily by the application for which the machine is intended. If the machine is for general-purpose data processing, then the basic arithmetic and logic operations must be included in the instruction set. Some machines such as the IBM 370 have separate instructions for binary and decimal (BCD) arithmetic, making them suitable for both scientific and business-oriented processing, respectively. Some processors (such as MOS Technology 6502 (M6502)) can operate in either binary or decimal mode; that is, the same instruction operates on either type of data. The logical operations AND, OR, NOT, and EXCLUSIVE-OR and the SHIFT and CIRCULATE operations are needed to access the data at bit and byte levels and to implement other arithmetic operations such as multiply and divide and floating-point arithmetic. Some control instructions for branching (conditional and unconditional), halt, and subroutine call and return are also required. Input/output (I/O) can be performed using dedicated instructions for input and output (INTEL 8080, IBM 370) or a memory-mapped I/O (DEC PDP-11, M6502) where the I/O devices are treated as memory locations and hence all the operations using memory are also applicable to I/O.

In general, an instruction set can be classified according to the type of operand used by each instruction. The operand is located either in a register or in a memory location. The typical classes of instructions are

1. Memory-to-memory instructions. Both operands are in memory.
2. Memory-to-register instructions. One of the operands is in memory, the other in a register.
3. Register-reference instructions. The operation is performed on the contents of one or more registers.
4. Memory-reference instructions. The operation is performed on the contents of memory locations.
5. Control instructions. These are branching, halt, pause, and the like.
6. Input/output instructions.
7. Macroinstructions. These are equivalent to a set of (more than one) instructions of the types 1 through 6.

Note that the arithmetic and logic instructions are a subset of the memory and register reference instructions.

As we have seen in Chapter 5, the complexity of the control unit increases as the number of instructions in the instruction set increases. Machines designed during the SSI and MSI (small- and medium-scale integration) era tended to have small instruction sets in order to limit the hardware complexity since the hardware was expensive. With the advent of very large-scale integrated-circuit (VLSI) technology, hardware complexity and cost were not the limiting factors; consequently, machines with large instruction sets and powerful instructions were designed, ushering in the era of *complex instruction set computers* (CISC). The instruction sets of even microcomputers in this era are as elaborate and as powerful as those of mainframes of the early seventies. In addition to complex processing functions, the instruction sets of these machines include high-level language constructs such as loop control and multiway branches as a single instruction rather than a series of assembly-language-level instructions.

A study of application programs in any language reveals that only a small set of instructions in a large instruction set are used frequently; hence, a small instruction set with the most often used instructions would serve equally well while limiting the hardware complexity. This observation has had a great effect on processor architectures because the silicon area on an IC chip saved by using a small instruction set can be used to implement more powerful hardware operators and greater word lengths. Such designs produce *reduced instruction set computers* (RISC). We will describe these architectures further in Chapters 8–11.

### 7.3.1 Instruction Length

The length of an instruction is a function of the number of addresses in the instruction. Typically, a certain number of bits are needed for the opcode; register references do not require a large number of bits, while memory references consume the major portion of an instruction. Hence, memory-reference and memory-to-memory instructions will be longer than the other types of instructions. A variable-length format can be used with these to conserve the amount of memory occupied by the program, but this increases the complexity of the control unit.

The number of memory addresses in an instruction dictates the speed of execution of instruction, in addition to increasing the instruction length. One memory access is needed to fetch the instruction from the memory if the complete instruction is in one memory word. If the instruction is longer than one word, multiple memory accesses are needed to fetch it unless the memory architecture provides for the access of multiple words simultaneously (as described in Chapter 8). Index and indirect address computation are needed for each memory address in the instruction, thereby adding to the instruction processing time. During the instruction execution phase, corresponding to each memory address a memory read or write access is needed. Since the memory access is slower than a register transfer, the instruction processing time is considerably reduced if the number of memory addresses in the instruction are kept to a minimum.

Based on the number of addresses (operands), the following instruction organizations can be envisioned:

1. Three-address
2. Two-address
3. One-address
4. Zero-address

We now compare the above organizations using the typical instruction set required to accomplish the four basic arithmetic operations. In this comparison, we assume that the instruction fetch requires one memory access in all cases and that, since all the addresses are direct addresses, the address computation does not require any memory accesses. Therefore, we will compare only the memory accesses needed during the execution phase of the instruction.

#### Three-address Machine

| | | |
|---|---|---|
| ADD | A, B, C | $M[C] \leftarrow M[A] + M[B]$ |
| SUB | A, B, C | $M[C] \leftarrow M[A] - M[B]$ |

| | | |
|---|---|---|
| MPY | A, B, C | M[C] ← M[A] · M[B] |
| DIV | A, B, C | M[C] ← M[A] / M[B] |

A, B, and C are memory locations.

Each of the above instructions requires three memory accesses during execution. In practice, the majority of operations are based on two operands with the result occupying the position of one of the operands. Thus, the instruction length and address computation time can be reduced by using a two-address format, although three memory accesses are still needed during the execution.

### Two-address Machine

| | | |
|---|---|---|
| ADD | A, B | M[A] ← M[A] + M[B] |
| SUB | A, B | M[A] ← M[A] − M[B] |
| MPY | A, B | M[A] ← M[A] · M[B] |
| DIV | A, B | M[A] ← M[A] / M[B] |

The first operand is lost after the operation.

If one of the operands can be retained in a register, the execution speeds of the above instructions can be increased. Further, if the operand register is implied by the opcode, a second operand field is not required in the instruction, thus reducing the instruction length:

### One-address Machine

| | | |
|---|---|---|
| ADD | A | ACC ← ACC + M[A] |
| SUB | A | ACC ← ACC − M[A] |
| MPY | A | ACC ← ACC · M[A] |
| DIV | A | ACC ← ACC / M[A] |
| LOAD | A | ACC ← M[A] |
| STORE | A | M[A] ← ACC |

"ACC" is an accumulator or any other register implied by the instruction. (An accumulator is required here.) If both the operands can be held in registers, the execution time is decreased. Further, if the opcode implies the two registers, instructions can be of zero-address type:

### Zero-address Machine

| | |
|---|---|
| ADD | SL ← SL + TL, POP |
| SUB | SL ← SL − TL, POP |

```
MPY          SL ← SL · TL, POP
DIV          SL ← SL / TL, POP
LOAD    A    PUSH, TL ← M[A]
STORE   A    M[A] ← TL, POP
```

SL and TL are the second- and top-level respectively of a last-in, first-out stack. In a zero-address machine, all the arithmetic operations are performed on the top two levels of a stack, so no explicit addresses are needed as part of the instruction. However, some memory reference (one-address) instructions such as LOAD and STORE (or MOVE) to move the data between the stack and the memory are required.

Appendix D describes the two most popular implementations of stack. If the zero-address machine uses a hardwired stack, then the above arithmetic instructions do not require any memory access during execution; if a RAM-based stack is used, each arithmetic instruction requires three memory accesses.

Assuming $n$ bits for an address representation and $m$ bits for the opcode, the instruction lengths in the above four organizations are:

Three-address: $m + 3n$ bits
Two-address: $m + 2n$ bits
One-address: $m + n$ bits
Zero-address: $m$ bits.

Figure 7.1 offers programs to compute the function F = A · B + C · D using each of the above instruction sets. Here A, B, C, D, and F are memory locations. Contents of A, B, C, and D are assumed to be integer values, and the results are assumed to fit in one memory word. Program sizes can be easily computed from benchmark programs of this type, using the typical set of operations performed in an application environment. The number of memory accesses needed for each program are also shown in the figure. Although these numbers provide a measure of relative execution times, the time required for other (non-memory access) operations should be added to these times to complete the execution time analysis. Results of such benchmark studies are used in the selection of instruction sets and instruction formats and comparison of processor architectures.

In IBM 370, the instructions are 2, 4 or 6 bytes long; DEC PDP-11 employs an innovative addressing scheme to represent both single and double operand instructions in two bytes. An instruction in INTEL 8080 is either 1, 2, or 3 bytes long. The instruction formats are discussed later in this section.

| Program | Program length (bits) | Number of Memory Accesses | |
|---|---|---|---|
| | | FETCH | EXECUTE |
| Three-address | | | |
| MPY A, B, A<br>MPY C, D, C<br>ADD A, C, F | $3(m + 3n)$ | 3 | $3 \cdot 3 = 9$ |
| Two-address | | | |
| MPY A, B<br>MPY C, D<br>ADD A, C<br>SUB F, F<br>ADD F, A | $5(m + 2n)$ | 5 | $5 \cdot 3 = 15$ |
| One-address | | | |
| LOAD A<br>MPY B<br>STORE F<br>LOAD C<br>MPY D<br>ADD F<br>STORE F | $7(m + n)$ | 7 | $7 \cdot 1 = 7$ |
| Zero-address | | | |
| LOAD A<br>LOAD B<br>MPY<br>LOAD C<br>LOAD D<br>MPY<br>ADD<br>STORE F | $3m + 5(m + n)$ | 8 | $5 \cdot 1 = 5$<br>(Assuming hardware stack) |

**Figure 7.1** Programs to compute $F = A \cdot B + C \cdot D$

### 7.3.2 Opcode Selection

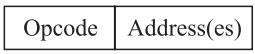
Assignment of opcode for instructions in an instruction set can significantly influence the efficiency of the decoding process at the execute phase of the instruction cycle. (ASC opcodes have been arbitrarily assigned.) Two opcode assignments are generally followed:

1. Reserved opcode method
2. Class code method.

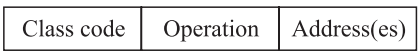
In the reserved opcode method, each instruction would have its own opcode. This method is suitable for instruction sets with fewer instructions. In the

class code method, the opcode consists of two parts: a class code part and an operation part. The class code identifies the type or class of instruction and the remaining bits in the opcode (operation part) identify a particular operation in that class. This method is suitable for larger instruction sets and for instruction sets with variable instruction lengths. Class codes provide a convenient means of distinguishing between various classes of instructions in the instruction set. The two opcode assignment modes are illustrated here:

Reserved opcode instruction.




Class code instruction.




In practice, it may not be possible to identify a class code pattern in an instruction set. When the instructions are of fixed length and the bits of the opcode always completely decoded, there may not be any advantage to assigning opcodes in a class code form. For example, the INTEL 8080 instruction set does not exhibit a class code form; in IBM 370, the first 2 bits of the opcode distinguish between 2-, 4-, and 6-byte instructions.

M6502, an 8-bit microprocessor, uses a class code in the sense that part of the opcode distinguishes between the allowed addressing modes of the same instruction. For example, the "add memory to accumulator with carry (ADC)" instruction has the following opcode variations:

| Addressing mode | Opcode (H) |
|---|---|
| Immediate mode | 69 |
| Zero page | 65 |
| Zero page, index X | 75 |
| Nonzero page | 6D |
| Nonzero page, index X | 7D |
| Nonzero page, index Y | 79 |
| Preindexed-indirect | 61 |
| Postindexed-indirect | 71 |

We will describe the paged addressing mode later in this chapter.

### 7.3.3 Instruction Formats

The majority of machines use a fixed-field format within the instruction while varying the length of the instruction to accommodate a varying number of addresses. Some machines vary even the field format within an instruction. In such cases, a specific field of the instruction defines the instruction format.

Figure 7.2 shows the instruction formats of the Motorola 6800 and 68000 series of machines. The MC6800 is an 8-bit machine and can directly address 64 KB of memory. Instructions are one, two, or three bytes long. Members of the MC68000 series of processors have an internal 32-bit (data- and address-) architecture. The MC68000 provides a 16-bit data bus and a 24-bit address bus, while the top-of-the-line MC68020 provides 32-bit data and 32-bit address buses. The instruction format varies from register-mode instructions that are one (16-bit) word long to memory reference instructions that can be two to five words long.

Figure 7.3 shows the instruction formats for DEC PDP-11 and VAX-11 series. PDP-11 is a 16-bit processor series. In its instruction format, R indicates one of the eight general-purpose registers, I is the direct/indirect

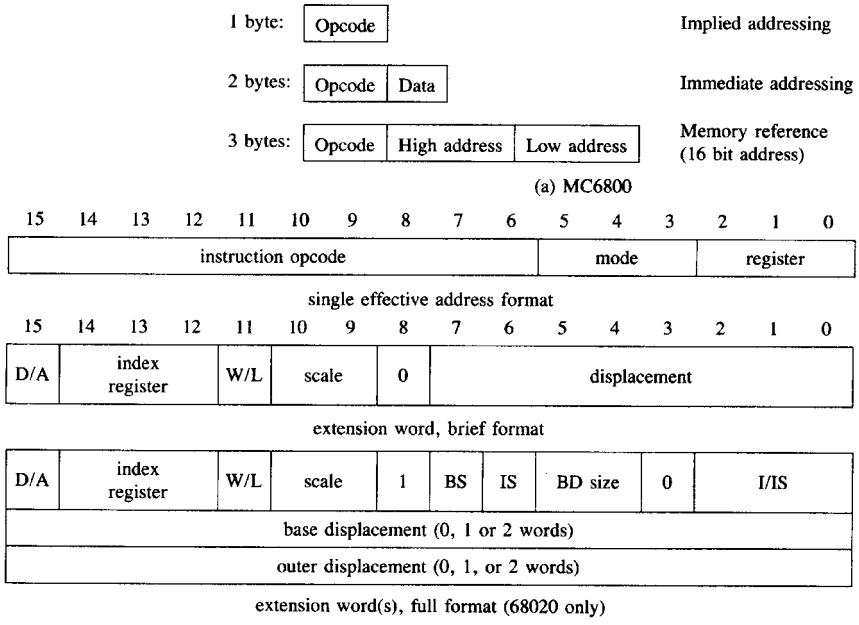



**Figure 7.2** Instruction formats (*continues*)

| | |
|---|---|
| I/IS | Index/Indirect select (68020 only) |
| BD size | Base displacement size (68020 only)<br>00 Reserved<br>01 Null displacement<br>10 Word displacement<br>11 Long displacement |
| IS | Index suppress (68020 only) |
| BS | Base suppress (68020 only)<br>0 Evaluate and add base register<br>1 Suppress base register |
| W/L | Index register size<br>0 Sign-extended word<br>1 Signed long word |
| Index register | Data or address register |
| D/A | Index register type<br>0 Data register<br>1 Address register |
| register | Data or address register |
| mode | Addressing mode |
| opcode | Instruction and possible mode/register information for second operand |
| displacement | Signed 8-bit value |
| scale | Index scaling factor (68020 only)<br>00 = 1X<br>01 = 2X<br>10 = 4X<br>11 = 8X |

(b) MC68000

**Figure 7.2** (Continued)

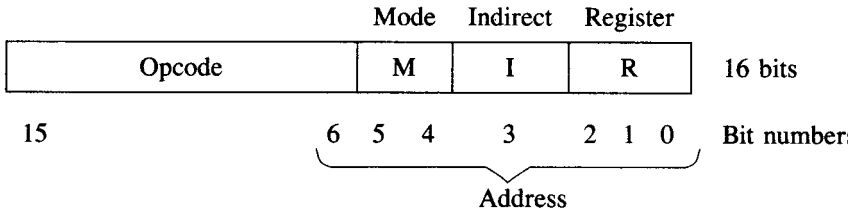
flag, and M indicates the mode in which the register is used. The registers can be used in operand, autoincrement pointer, autodecrement pointer, or index modes. We will describe these modes later in this chapter. The VAX-11 series of 32-bit processors have a very generalized instruction format. An instruction consists of the opcode (1 or 2 bytes long) followed by from 0 to 6 operand specifiers. An operand specifier is between 1 to 8 bytes long. Thus a VAX-11 instruction can be from 1 to 50 bytes long.

Figure 7.4 shows the instruction formats of IBM 370, a 32-bit processor whose instructions can be from 2 to 6 bytes long. Here, R1, R2 and R3 refer to one of the 16 general-purpose registers (GPRs), B1 and B2 are *base* register designations (one of the GPRs), DATA indicates immediate data, D1 and D2 refer to a 12-bit *displacement*, and L1 and L2 indicate the *length* of data in bytes. The base-displacement addressing mode is described later in this chapter.

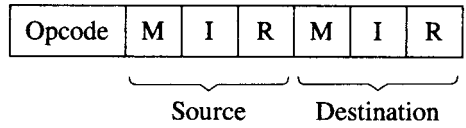
## 7.4 REGISTERS

Each of the registers in ASC is designed to be a special-purpose register (accumulator, index register, program counter, etc.). The assignment of functions limits the utility of these registers, but such machines are simpler

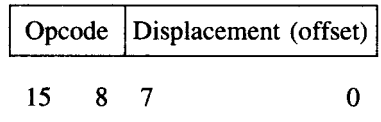
Single-operand instruction.



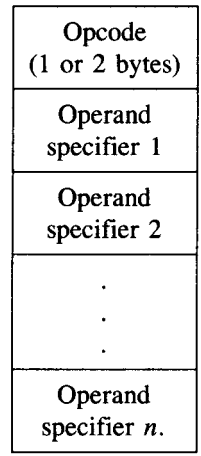

Double-operand instruction




Branch instruction




(a) PDP-11




(b) VAX-11

**Figure 7.3** Instruction formats

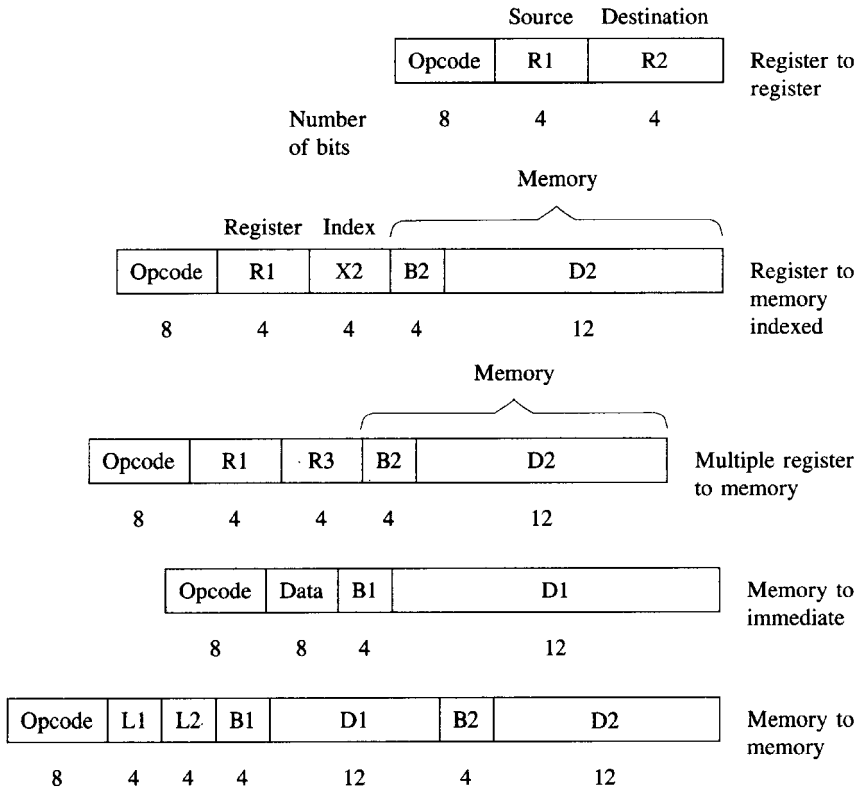
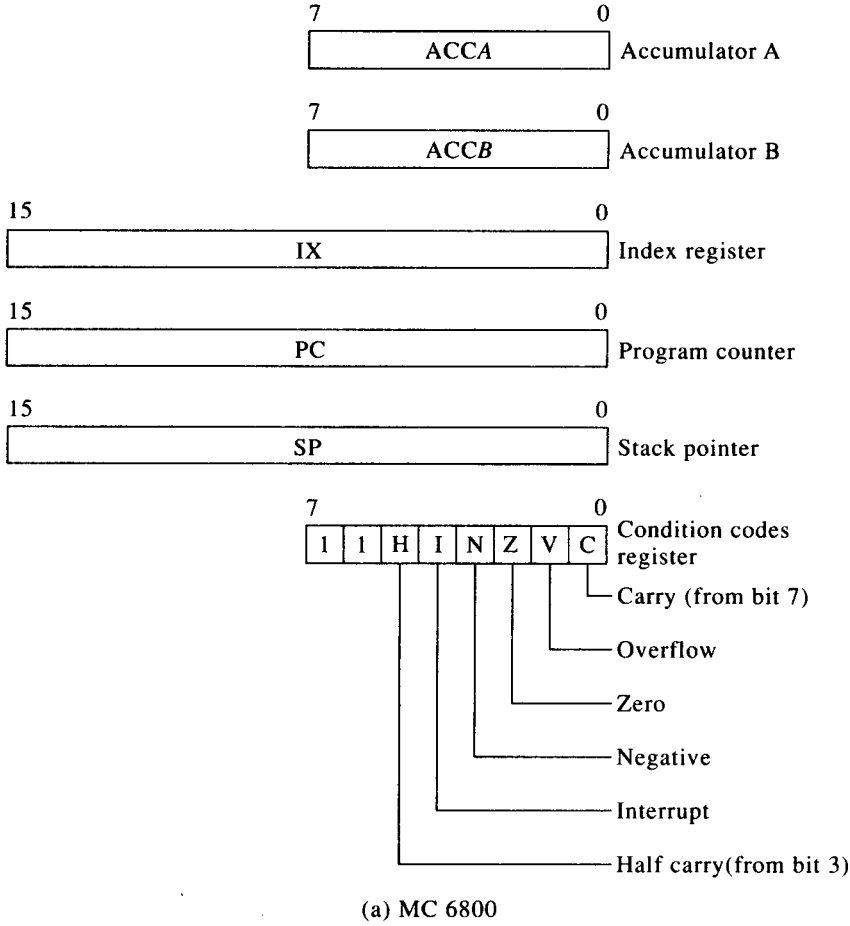



**Figure 7.4** IBM 370 instruction formats

to design. As the hardware technology has advanced, yielding lower cost hardware, the number of registers in the CPU has increased. The majority of registers are now designated as general-purpose registers and hence can be used as accumulators, index registers, or pointer registers (that is, registers whose content is an address pointing to a memory location). The *processor status register* is variously referred to as the status register, condition code register, program status word, etc.

Figure 7.5 shows the register structures of MC6800 and MC68000 series processors. The MC68000 series of processors operate in either *user* or *supervisory* mode. Only user mode registers common to all the members in the processor series are shown. In addition, these processors have several supervisor mode registers. The type and number of supervisor mode registers vary among the individual processors in the series. For example, the MC68000 has a system stack pointer and uses 5 bits in the most

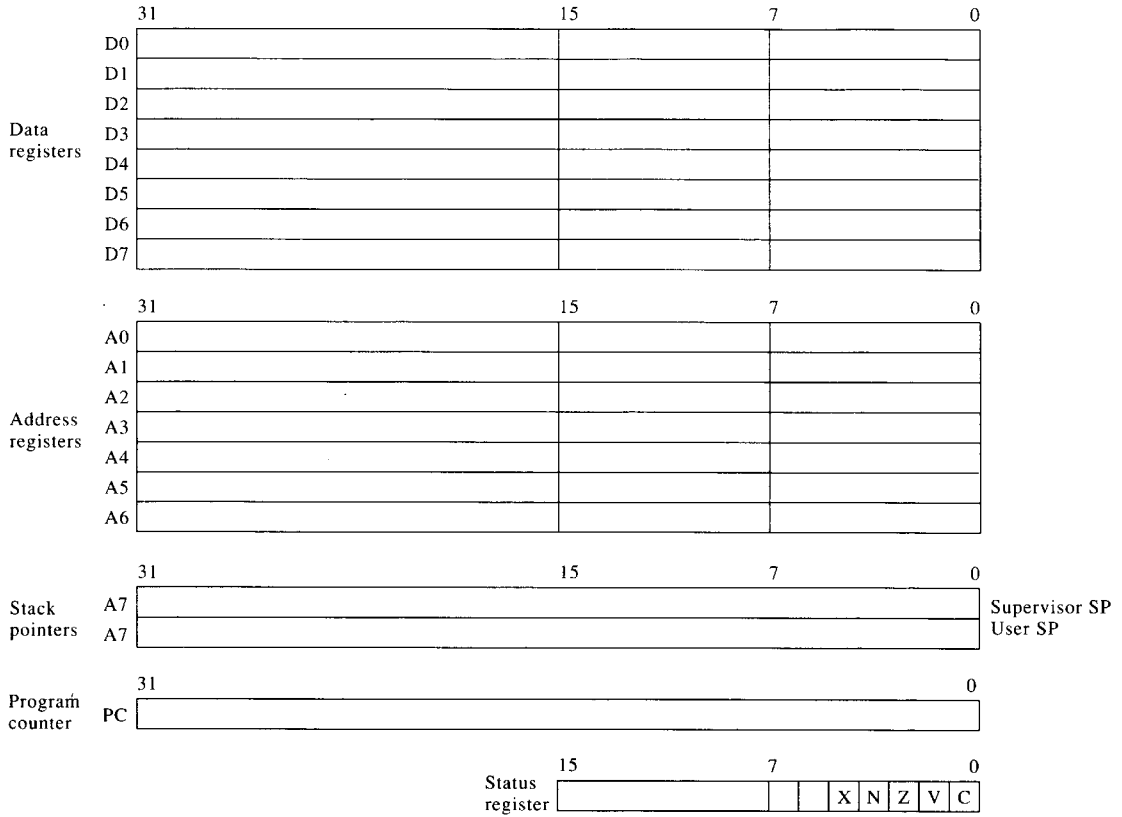



(a) MC 6800

**Figure 7.5** Register structure *(continued)*

significant byte of the status register while the MC68020 has 7 additional registers.

Figure 7.6 shows the register structures of INTEL 8080, 8086, and 80386 processors. Note that the same set of general-purpose registers (A, B, C, and D) is maintained, except that their width has increased from 8 bits in the 8080 to 32 bits in the 80386. These registers also handle special functions for certain instructions. For instance, C is used as a counter for loop control and to maintain the number of shifts during shift operations, and B is used as a base register. The *source* (SI) and *destination* (DI) index registers are used in string manipulation.




(b) MC 68000

**Figure 7.5** (Continued)

Register pairs B, C, and D, E—each register 8 bits long; 8-bit operands utilize individual registers and 16-bit operands utilize register pairs.
Register pair H, L—usually used for address storage.
Stack pointer—16 bits.
Program counter—16 bits.
Accumulator—8 bits.
Buffer registers to store ALU operands—two, 8 bits long.
Status flag register:

| Z | C | S | P | AC |
|---|---|---|---|---|

5 bits.

Z is zero, C is carry, S is sign, P is parity, and AC is auxiliary carry (for decimal arithmetic).

(a) 8080 (See Fig. 7.13)

**8-bit data registers** (AH, AL; BH, BL; CH, CL; DH, DL), Each pair can be used as a 16-bit register. The pairs are designated AX, BX, CX, and DX.

**16-bit registers:**

SP: Stack pointer BP: Base pointer DI: Destination index
SI: Source index IP: Instruction pointer (PC)

Segment registers: Code (CS), Data (DS), Stack (SS), Extra (ES)

Status register: Containing Overflow, Direction, Interrupt, Trap, Sign, Zero, Auxiliary carry, Parity, Carry flags.

(b) 8086 (See Fig. 7.14)

**32-bit registers**

General data and address registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

Instruction pointer: EIP

Flag register: EFLAGS

(All are extended versions of corresponding 8086 registers)

**16-bit registers**

Segment registers: CS, SS, DS, ES, FS, and GS (The last four are used in data addressing)

(c) 80386

**Figure 7.6** Register structures of INTEL microprocessors

INTEL 8086 series views the memory as consisting of several *segments*. There are four segment registers. The code segment (CS) register points to the beginning of code (i.e., program) segment; the *data segment* (DS) register points to the beginning of data segment; the *stack segment* (SS) points to the beginning of stack segment, and the *extra segment* (ES) register points to that of the extra (data) segment. The *instruction pointer* (IP) is the program counter, and its value is a displacement from the address pointed to by CS. We will discuss this base/displacement addressing mode later in this chapter. Similarly, the *stack pointer* (SP) contains a displacement value from the SS contents and points to the top level of the stack. The base pointer (BP) is used to access stack frames that are non-top stack levels. Refer to Section 7.6 for details on the Intel Pentium series of processors.

DEC PDP-I1 can operate in three modes: user, supervisory, and kernel. All instructions are valid in kernel mode, but certain instructions (such as HALT) are not allowed in the other two modes. There are two sets of six registers each (R0-R5): set 0 is for the user mode and set 1 is for the other modes. There are three stack pointers, one for each mode (R6) and one program counter. All registers can be used in operand (contain data), pointer (contain address), and indexed modes. The processor status word format is shown in Fig. 7.7

DEC VAX-11 maintains the general register structure of PDP-11 and contains sixteen 32-bit GPRs, a 32-bit processor status register, four 32-bit registers used with system console operations, four 32-bit clock function and timing registers, and a 32-bit floating-point accelerator control register.

IBM 370 is a 32-bit machine. There are sixteen general-purpose registers, each 32 bits long. They can be used as base registers, index registers, or operand registers. There are four 64-bit floating-point registers. The program status word is shown in Fig. 7.8.

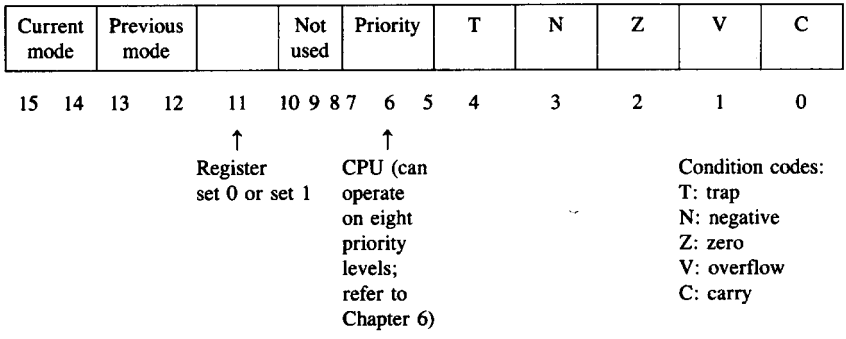



**Figure 7.7** PDP-11 processor status word

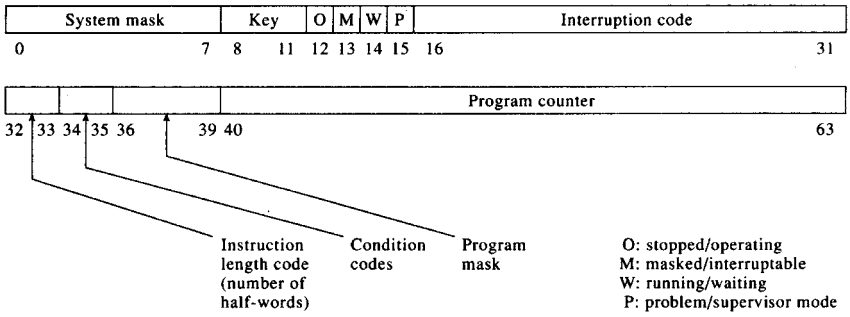



**Figure 7.8** IBM 370 processor status word

## 7.5 ADDRESSING MODES

The direct, indirect, and indexed addressing modes are the most common. Some machines allow preindexed-indirect, some allow postindexed-indirect, and some allow both modes. These modes were described in Chapter 4. The other popular addressing modes are described below. In practice, processors adopt whatever combination of these popular addressing modes that suits the architecture.

### 7.5.1 Immediate Addressing

In this mode, the operand is a part of the instruction. The address field is used to represent the operand itself rather than the address of the operand. From a programmer's point of view, this mode is equivalent to the literal addressing used in ASC. The literal addressing was converted into a direct address by the ASC assembler. In practice, to accommodate an immediate addressing mode, the instruction contains a data field and an opcode. This addressing mode makes operations with constant operands faster since the operand can be accessed without another memory fetch.

### 7.5.2 Paged Addressing

When this mode of addressing is allowed, the memory is assumed to consist of several equal-sized blocks, or *pages*. The memory address can then be treated as having a page number and a location address (offset) within the page. Figure 7.9 shows a paged memory in which each page is 256 bytes long. The instruction format in such an environment will have a *page bit* and enough additional bits to address the largest offset (that is, the size of the

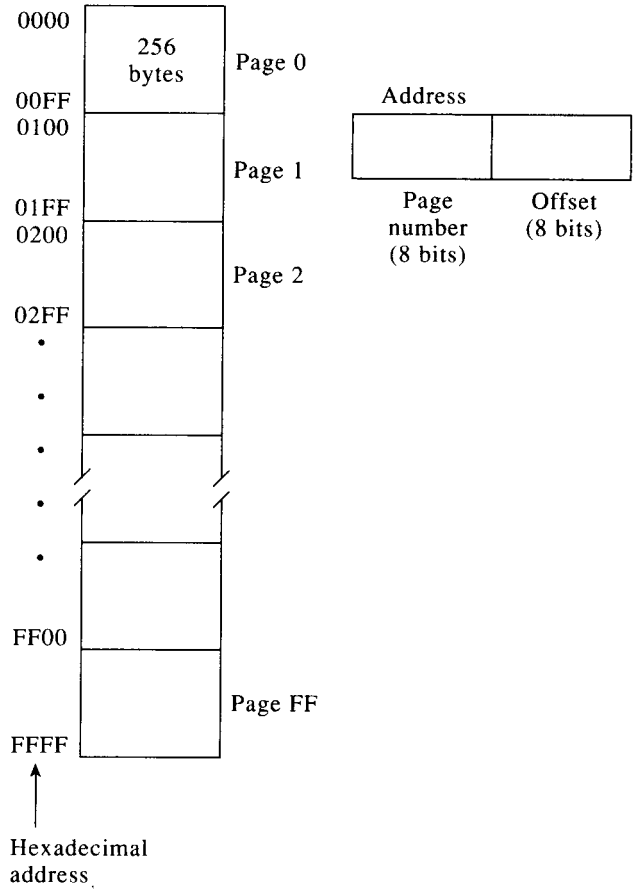
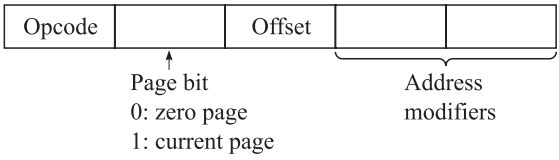



**Figure 7.9** Paged memory

page). Usually the "zero page" of the memory is used for the storage of the most often used data and pointers. The page bit specifies whether the address corresponds to the zero page or the "current page" (i.e., the page in which the instruction is located). Address modifiers (to indicate indexing, indirect, etc.) are also included in the instruction format, as shown below:



If the pages are large enough, the majority of the memory references by the program will be within the same page, and those locations are addressed with fewer bits in the address field than in the case of the direct addressing scheme. If the referenced location is beyond the page, an extra memory cycle may be needed to access it.

M6502 uses such an addressing scheme. The 16-bit address is divided into an 8-bit page address and an 8-bit offset within the page (of 256 bytes). Further, the processor has unique opcodes for zero-page mode instructions. These instructions are assembled into 2 bytes, the higher byte of the address being zero, while the non-zero page instructions need 3 bytes to represent the opcode and a 16-bit address.

Note that the page number can be maintained in a register; thus the instruction contains only the offset part of the address and a reference to the register containing the page number. The segment registers of INTEL 8086 essentially serve this purpose. The base-register addressing mode described next is a variation of paged addressing mode.

Paged memory schemes are useful in organizing virtual memory systems, as described in Chapter 8.

### 7.5.3 Base-register Addressing

In some machines, one of the CPU registers is used as a *base register*. The beginning address of the program is loaded into this register as a first step in program execution. Each address referenced in the program is an offset (displacement) with respect to the contents of the base register. Only the base register identification and the displacement are represented in the instruction format, thus conserving bits. Since the set of instructions to load the base register is part of the program, relocation of the programs is automatic. IBM 370 series machines use this scheme; any of the general-purpose registers can be designated as a base register by the programmer. The segment mode of addressing used by INTEL 8086 series is another example of base-register addressing.

### 7.5.4 Relative Addressing

In this mode of addressing, an offset (or displacement) is provided as part of the instruction. This offset is added to the current value of PC during execution to find an effective address. The offset can be either positive or negative. Such addressing is usually used for branch instructions. Since the jumps are usually within a few locations of the current address, the offset can be a small number compared to the actual jump address, thus reducing

the bits in the instruction. DEC PDP-11 and M6502 use such an addressing scheme.

### 7.5.5 Implied (Implicit) Addressing

In zero-address instructions, the opcode implies that the operands are on the top two levels of the stack. In one-address instructions, one of the operands is implied to be in a register similar to the accumulator.

Implementation of the addressing modes varies from processor to processor. Addressing modes of some practical machines are listed below.

#### M6502

M6502 allows direct, preindexed-indirect, and postindexed-indirect addressing. Zero-page addressing is employed to reduce the instruction length by 1 byte. The addresses are 2 bytes long. A typical absolute address instruction uses 3 bytes. A zero-page address instruction uses only 2 bytes.

#### MC6800

The addressing modes on MC6800 are similar to those of M6502 except that the indirect addressing mode is not allowed.

#### MC68000

This processor allows 14 fundamental addressing modes that can be generalized into the six categories shown in Fig. 7.10

#### INTEL 8080

The majority of the memory references for INTEL 8080 are based on the contents of register pair H, L. These registers are loaded, incremented, and decremented under program control. The instructions are thus only 1 byte long and imply indirect addressing through the H, L register pair. Some instructions have the 2-byte address as part of them (direct addressing). There are also single-byte instructions operating in an implied addressing mode.

| *Mode* | *Effective address* |
|---|---|
| Register direct addressing | |
| Address register direct | EA = A*n* |
| Data register direct | EA = D*n* |
| Absolute data addressing | |
| Absolute short | EA = {Next word} |
| Absolute long | EA = {Long words} |
| Program counter relative addressing | |
| Relative with offset | EA = {PC} + *d*16 |
| Relative with index and offset | EA = {PC} + {R*n*} + *d*8 |
| Register indirect addressing | |
| Register indirect | EA = {A*n*} |
| Postincrement register indirect | EA = {A*n*}, A*n* ← A*n* + N |
| Predecrement register indirect | A*n* ← A*n* − N, EA = {A*n*} |
| Register indirect with offset | EA = {A*n*} + *d*16 |
| Indexed register indirect with offset | EA = {A*n*} + {R*n*} + *d*8 |
| Immediate data addressing | |
| Immediate | Data = Next word(s) |
| Quick immediate | Inherent data |
| Implicit addressing | |
| Implicit register | EA = SR, USP, SP, PC |

EA: Effective address
{ }: Contents of
D*n*: Data register
A*n*: Address register
d*n*: *n*-bit offset
R*n*: Data or address register

**Figure 7.10** MC68000 addressing modes

## INTEL 8086

The segment-register-based addressing scheme of this series of processors is a major architectural change from the 8080 series. This addressing scheme makes the architecture flexible, especially for the development of compilers and operating systems.

## DEC PDP-11

The two instruction formats of PDP-11 shown earlier in this chapter provide a versatile addressing capability. Mode (M) bits allow the contents of the referenced register to be (1) an operand, (2) a pointer to a memory location that is incremented (auto increment) or decremented (auto decrement) automatically after accessing that memory location, or (3) an index. In the index mode the instructions are two words long. The content of the second word is an address and is indexed by the referenced register. The indirect bit (I) allows direct and indirect addressing in all the four register modes. In addition, PDP-11 allows PC-relative addressing, in which an offset is provided as part of the instruction and added to the current PC value to find the effective address of the operand. Some examples are shown below:

| Mnemonic | Octal opcode | Comments |
|---|---|---|
| CLR | 0050 *nn* | Clear word, *nn* is the register reference. |
| CLRB | 1050 *nn* | Clear byte. |
| ADD | 06 *nn mm* | Add, *nn* = source, *mm* = destination. |
| ADD R2, R4 | 06 02 04 | R4 ← R2 + R4. |
| CLR (R5) + | 0050 25 | Auto increment pointer, clear R5 ← 0, R5 ← R5 + 1. |
| ADD @X(R2), R1 | 06 72 01<br>Address | @ indicates indirect; X indicates indexing.<br>R1 ← M[M[Address + R2]] + R1 |
| ADD #10, R0 | 062700<br>000010 | immediate.<br>R0 ← 10. |
| INC Offset | 005267<br>Offset | M[PC + Offset] ← M[PC + Offset] + 1. |

### DEC VAX-11

The basic addressing philosophy of VAX-11 follows that of PDP-11 except that the addressing modes are much more generalized, as implied by the instruction format shown earlier in this chapter.

### IBM 370

Any of the 16 general-purpose registers can be used as an index register, an operand register, or a base register. The 12-bit displacement field allows 4 KB of displacement from the contents of the base register. A new base register is needed if the reference exceeds the 4 KB range. Immediate addressing is allowed. In decimal arithmetic where the memory-to-memory operations are used (6-byte instructions), the lengths of the operands are also specified, thus allowing variable-length operands. Indirect addressing is not allowed.

## 7.6 EXAMPLE SYSTEMS

Practical CPUs are organized around either a single-bus or multiple-bus structure. As discussed earlier, the single-bus structure offers simplicity of hardware and uniform interconnections at the expense of speed, while the multiple-bus structures allow simultaneous operations on the structure but

require complex hardware. Selection of the bus structure is thus a compromise between hardware complexity and speed. The majority of modern-day microprocessors are organized around single-bus structures since implementation of a bus consumes a great deal of the silicon available on an IC.

The majority of CPUs use parallel buses. If faster operating speeds are not required, serial buses can be employed, as is usually the case with the processor ICs used to build calculators. The design goal in fabricating a calculator IC is to pack as many functions as possible into one IC. By using serial buses, silicon area on the IC is conserved and used for implementing other complex functions.

Figure 7.11 shows a comparison of three-, two- and single-bus schemes for ASC. Note that the ADD operation can be performed in one cycle using the three-bus structure. In the two-bus structure, the contents of the ACC are first brought into the BUFFER register in the ALU in one cycle, and the results are gated into ACC in the second cycle, thus requiring two cycles for addition. In the single-bus structure, the operands are transferred into ALU buffer registers BUFFER1 and BUFFER2 one at a time and the result is then transferred into the ACC during the third cycle.

We will now provide a brief description of the processor structures of some early processors (MOS Technology 6502–M6502; Motorola 6800–MC6800; Intel 8080; Intel 8086; and Digital Equipment PDP-11). Although these processors are now obsolete, they are included here because of their simplicity compared to their modern counterparts. We then provide a detailed description of Intel Pentium II processor. The architectural concepts required to understand all the features of this processor are not yet covered in the book. The reader will therefore have to revisit this section after studying those architectural concepts in subsequent chapters. We also provide a description of the system structure of the SP700 workstation from Compaq Computer Corporation, followed by that of two contemporary workstation architectures.

### 7.6.1 MOS Technology 6502

The M6502 structure, shown in Fig. 7.12, uses an 8-bit data bus and two 8-bit address buses (ADH for the highest order 8 bits and ADL for the lower-order 9 bits of the 16-bit address). Since the M6502 is an 8-bit machine with 16-bit addresses, all address manipulations are performed in two parts (high and low parts) each containing 8 bits. These two parts are stored in ABL and ABH buffers, which are connected to the 16-bit address bus. The CPU provides an 8-bit bidirectional data bus for external transfers.

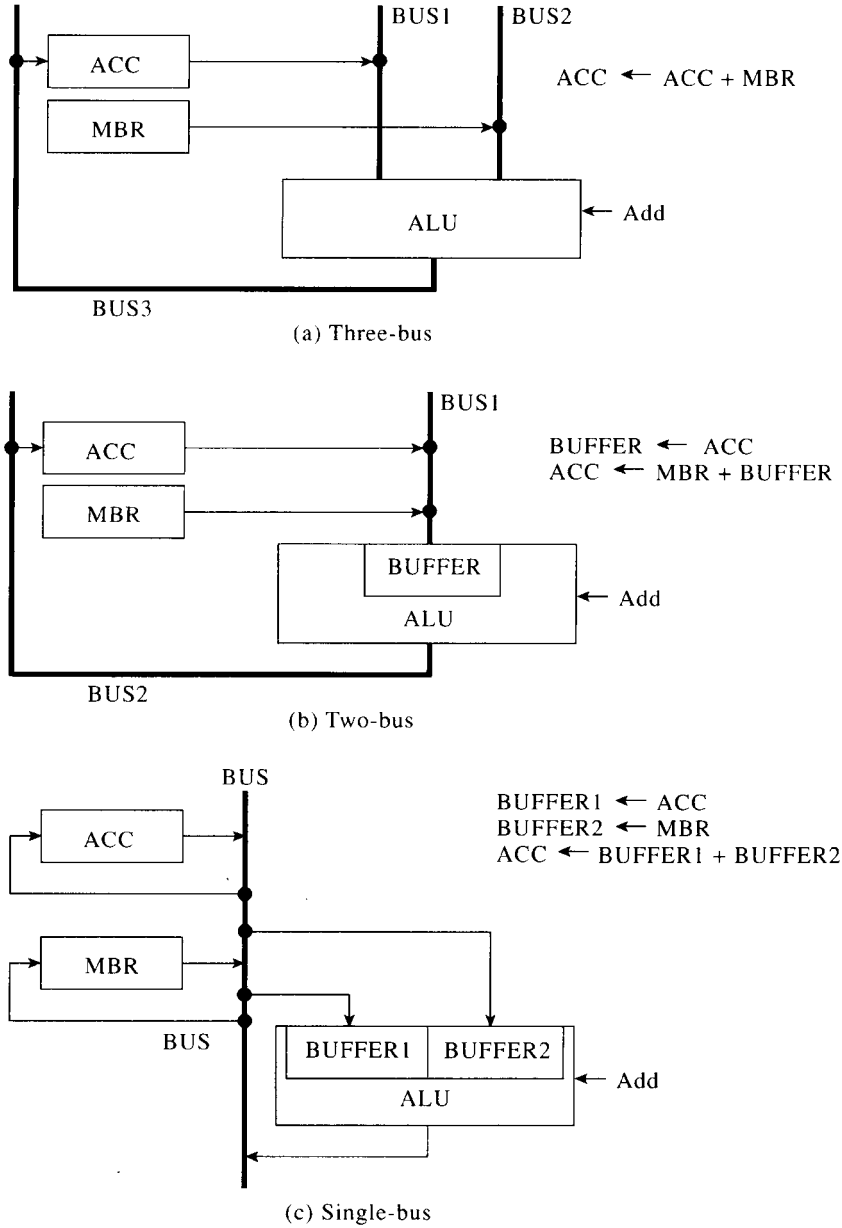



**Figure 7.11** Comparison of ASC bus architectures

### 7.6.2 Motorola 6800

The MC6800 uses a structure similar to that of M6502 except that it is configured around a single 8-bit bus rather than separate address and data buses.

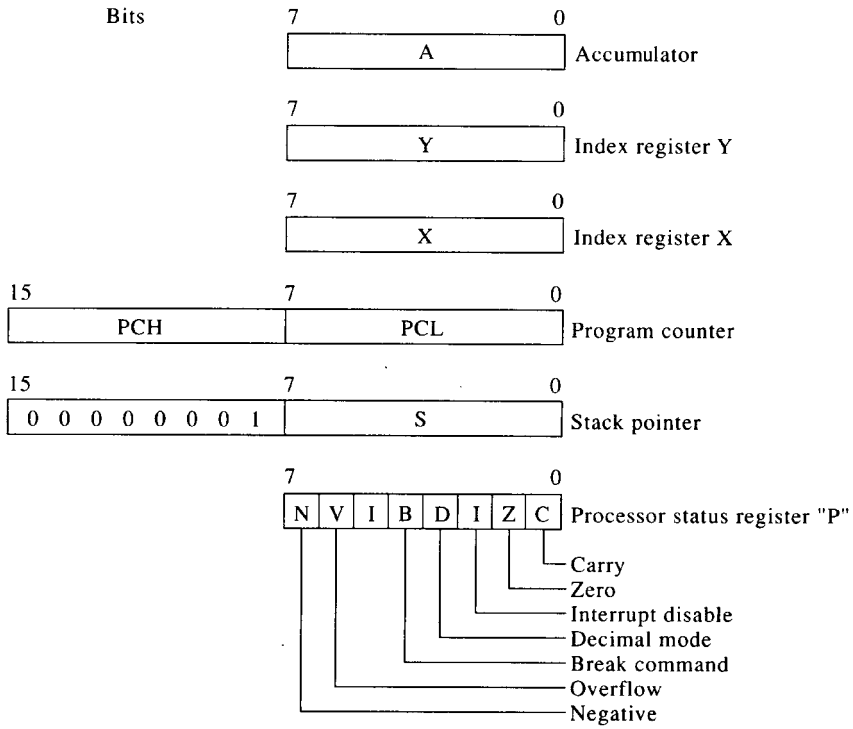
### 7.6.3 Intel 8080

The I8080 uses an 8-bit internal data bus for all register transfers (refer to Fig. 7.13). The arithmetic unit operates on two 8-bit operands residing in the temporary register (TEMP REG) and the accumulator latch. The CPU provides an 8-bit bidirectional data bus, a 16-bit address bus, and a set of control signals. These are used in building a complete microcomputer system.

All the above processors are single-bus architectures from the point of view of their ALUs. They all have buffers (such as the temporary register and accumulator latch of INTEL 8080) internal to their ALUs for storing the second operand during operations requiring two operands. These internal buffers are transparent to programmers.

### 7.6.4 Intel 8086

The I8086 processor structure shown in Fig. 7.14 consists of two parts: the execution unit (EU) and the bus interface unit (BIU). The EU is configured around a 16-bit ALU data bus. The BIU interfaces the processor with the external memory and peripheral devices via a 20-bit address bus and a 16-bit data bus. The EU and the BIU are connected via the 16-bit internal data bus and a 6-bit control bus (Q bus). EU and BIU are two independent units that work concurrently, thus increasing the instruction-processing speed. The BIU generates the instruction address and transfers instructions from the memory into an instruction buffer (instruction queue). This buffer can hold up to six instructions. The EU fetches instructions from the buffer and executes them. As long as the instructions are fetched from sequential locations, the instruction queue remains filled and the EU and BIU operations remain smooth. When the EU refers to an instruction address beyond the range of the instruction queue (as may happen during a jump instruction), the instruction queue needs to be refilled from the new address. The EU has to wait until the instruction is brought into the queue. Instruction buffers are further described in Chapter 9.




(a) Programming model

*(continues)*

**Figure 7.12** M6502 structure (Courtesy of MOS Technology Inc.)

### 7.6.5 Digital Equipment PDP-11

Figure 7.15 shows the structure of the PDP-11/45 CPU. All data paths within the CPU are 16 bits wide. There are two processor blocks (an arithmetic-logic processor and a floating-point processor), sixteen general-purpose registers, and a program status register. The memory management unit allows the use of solid-state (semiconductor) and core memory blocks simultaneously in the system. The CPU is configured as a device connected to the unibus (universal bus) along with memory and peripheral devices.

The unibus priority arbitration logic resolves the relative priorities of the devices requesting the bus. The CPU can be programmed to operate on eight levels of priority. Once the bus is granted, the requesting device becomes the bus master to transfer data to another device (slave). A second unibus (unibus B) is configured as a memory bus to which semiconductor memory blocks are connected.

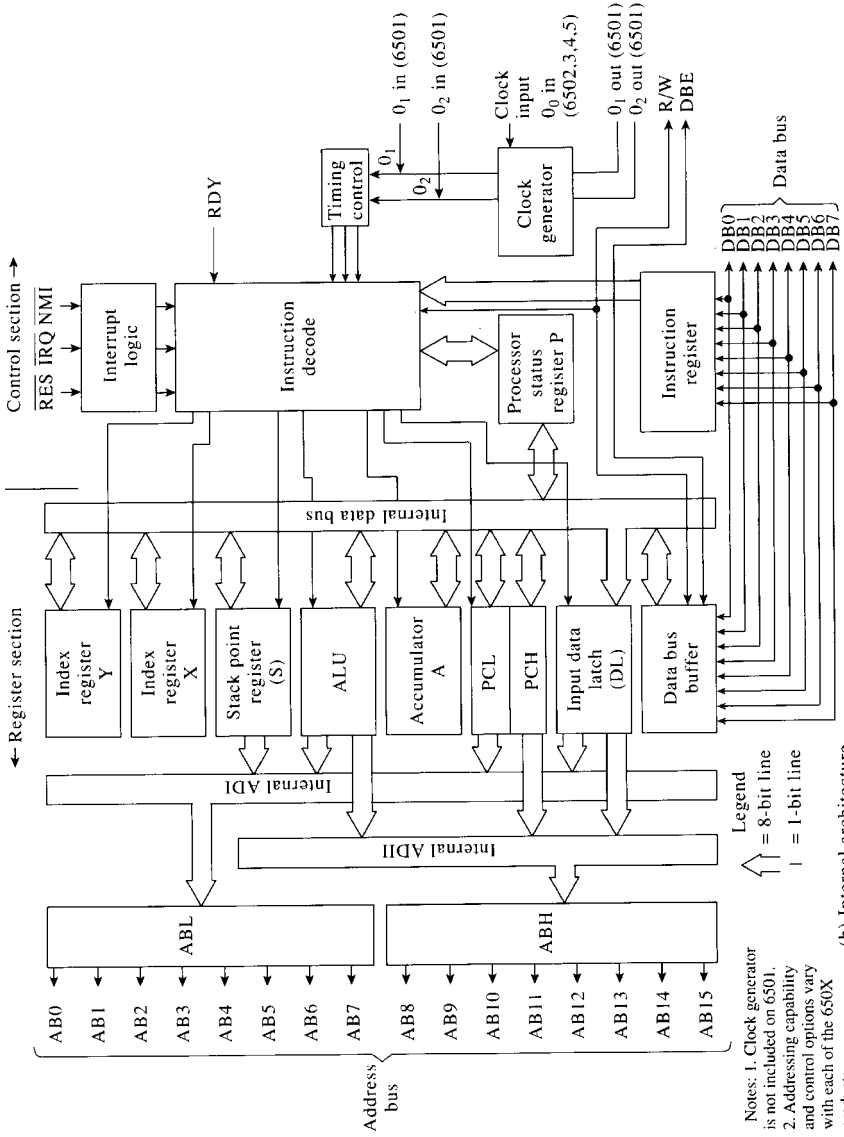



**Figure 7.12** (Continued)

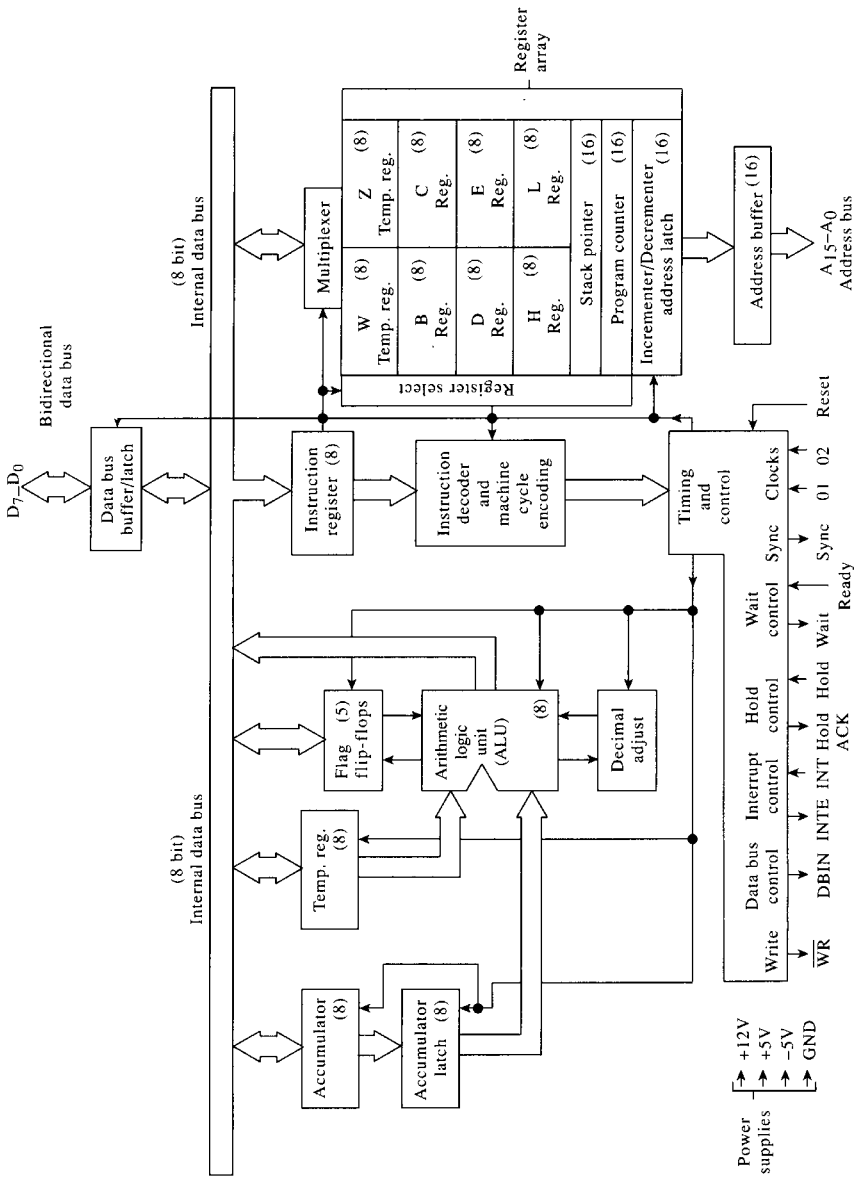



**Figure 7.13** INTEL 8080 functional block diagram (reprinted by permission of Intel Corporation, Copyright 1977. All mnemonics copyright Intel Corporation 1977)

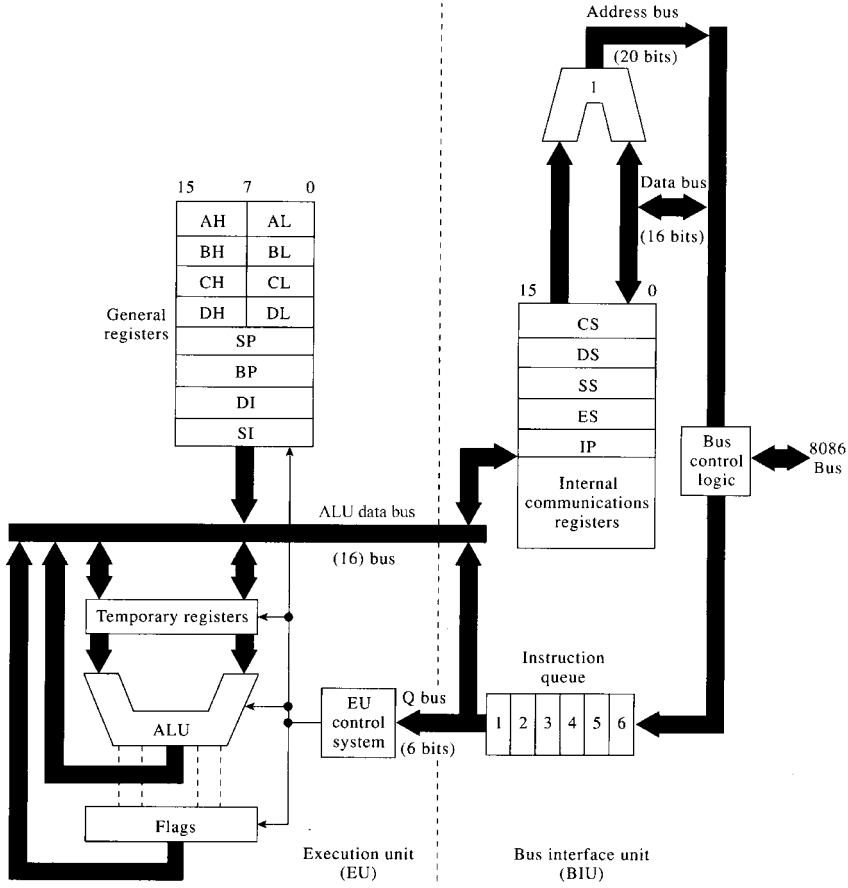



**Figure 7.14** INTEL 8086 elementary block diagram (reprinted by permission of Intel Corporation, copyright 1981. All mnemonics copyright Intel Corporation 1981)

As described in Chapter 6, the unibus is the communication path linking the CPU, memory, and peripheral devices. It is an asynchronous bus capable of transferring one 16-bit word every 400 ns. The unibus consists of the following 56 lines:

18 address lines (256K byte or 128K word address space)
16 data lines plus 2 parity lines
2 lines for type of transfer (data in, data out, data in pause, data-out byte)

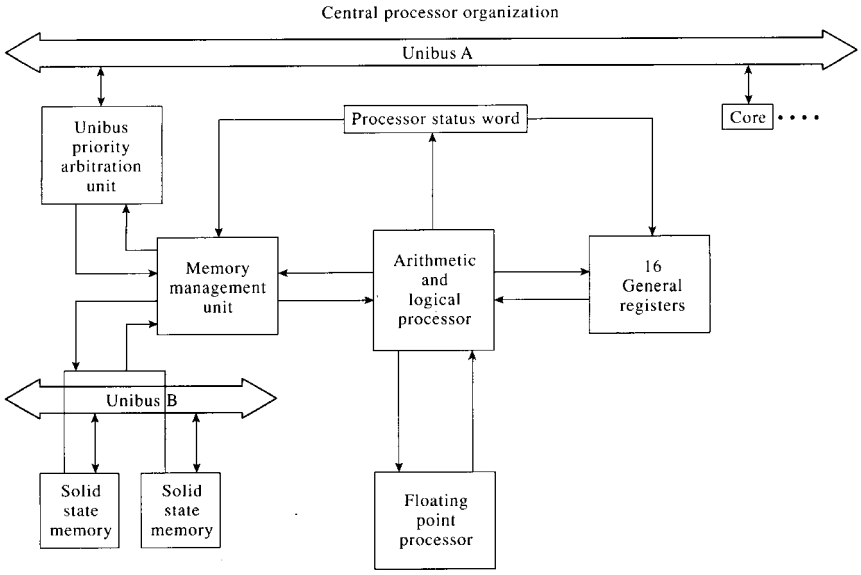



**Figure 7.15** PDP-11/45 structure (copyright, Digital Equipment Corporation, 1978. All Rights Reserved)

8 lines for bus control, timing, and status
5 bus request lines
5 unidirectional bus grant lines.

The unibus provides uniform interfacing characteristics for all devices, thus making interfacing easier.

### 7.6.6 Intel Pentium II

This section is extracted from Intel hardware and software developers manuals. We have provided a complete description of the Pentium II processor in this section, although some of the architectural features of this processor are not yet described in this book. They will be covered in subsequent chapters and hence the reader will have to revisit this section after gathering those concepts from later chapters.

The 4004 microprocessor was the first designed by Intel in 1969, followed by the 8080 and the 8085. The current Intel architecture is based on these original processors, but the architecture on which the Pentium II is based did not appear until the Intel 8086. The 8086 had a 20-bit address and could reference 1 megabyte (MB) of memory. The 8086 had 16-bit registers

and a 16-bit external data bus. The sister processor, the 8088, had an 8-bit bus. Memory was divided into 64 KB segments and four 16-bit "real mode" registers were used as pointers to addresses of the active segments. The Intel 80286 introduced protected mode, using the segment register as a pointer into address tables. The address table provided 24-bit addresses so that 16 MB of physical memory could be addressed and also provided support for virtual memory. Table 7.2 shows some of the features in each generation of the Intel Processor architecture.

The Intel 80386 included 32-bit registers to be used for operands and for addressing. The lower 16-bits of the new registers were used for compatibility with software written for earlier versions of the processor. The 32-bit addressing and 32-bit address bus allowed each segment to be as large as 4 GB. 32-bit operand and addressing instructions, bit manipulation instructions and paging were introduced in the 80386. The 80386 had six parallel stages for the bus interface, code pre-fetching, instruction decoding, execution, segment addressing and memory paging. The 80486 processor added more parallel execution capability by expanding the 80386 processor's instruction decode and execution units into five pipelined stages. Each stage could do its work on one instruction in one clock executing one instruction per CPU clock. An 8 KB on-chip L1 cache was added to the 80486 processor to increase the proportion of instructions that could execute at the rate of one per clock cycle. The 80486 included an on-chip floating point unit (FPU) to increase the performance.

The Intel Pentium processor (or the P6), introduced in 1993, added a second execution pipeline to achieve performance of executing two instructions per clock cycle. The L1 cache was increased to 8 KB devoted to instructions and another 8 KB for data. Write-back and write-through modes were included for efficiency. Branch prediction with an on-chip branch table was added to increase performance in pipeline performance for looping constructs.

The Intel Pentium Pro processor, introduced in 1995, included an additional pipeline so that the processor could execute three instructions per CPU clock. The Pentium Pro processor provides microarchitecture for flow analysis, out-of-order execution, branch prediction, and speculative execution to enhance pipeline performance. A 256 KB L2 cache that supports up to four concurrent accesses is included. The Pentium Pro processor also has a 36-bit address bus, giving a maximum physical address space of 64 GB.

The Intel Pentium II processor is an enhancement of the Pentium Pro architecture that combines functions of the P6 microarchitecture and included support for the Intel MMX$^{TM}$ Single Instruction Multiple Data (SIMD) processing technique and eight 64-bit integer MMX instruction

**Table 7.2** Intel Processor Feature History

| Intel processor | Date of product introduction | Performance (MIPs) | Max. CPU frequency at introduction (MHz) | No. of transistors on the die | Main CPU register size (bits) | Extern. data bus size | Max. extern. addr. space | Caches in CPU package |
|---|---|---|---|---|---|---|---|---|
| 8086 | 1978 | 0.8 | 8 | 29 K | 16 | 16 | 1 MB | None |
| Intel 286 | 1982 | 2.7 | 12.5 | 134 K | 16 | 16 | 16 MB | |
| Intel386™ DX | 1985 | 6.0 | 20 | 275 K | 32 | 32 | 4 GB | |
| Intel486™ DX | 1989 | 20 | 25 | 1.2 M | 32 | 32 | 4 GB | 8 KB L1 |
| Pentium ® | 1993 | 100 | 60 | 3.1 M | 32 | 64 | 4 GB | 16 KB L1 |
| Pentium Pro | 1995 | 440 | 200 | 5.5 M | 32 | 64 | 64 GB | 16 KB L1; 256 KB or 512 KB L2 |

registers for use in multimedia and communication processing. The Pentium II is available with clock speeds from 233 to 450 MHz.

The Pentium II processor can directly address $2^{32} - 1$ or 4 GB of memory on its address bus. Memory is organized into 8-bit bytes. Each bytes is assigned a physical memory location, called a physical address. To provide access into memory, the Pentium II provides three types of addressing schemes: flat, segmented and real-addressing modes. The flat addressing mode provides a linear address space. Program instructions, data and stack contents are present in a sequential byte order. Only 4 GB of byte-ordered memory is supported in flat addressing. The segmented memory model provides independent memory address spaces that are typically used for instructions, data and stack implementations. The program issues a logical address, which is a combination of the segment selector and an address offset, which identifies a particular byte in the segment of interest. $2^{16} - 1$ or 16, 232 segments of a maximum of $2^{32} - 1$ or 4 GB are addressable by the Pentium II providing a total of approximately $2^{48}$ or 64 terabytes (TB) addressable memory. The real-addressing mode is provided to maintain the Intel 8086 based memory organization. This is provided for backward compatibility for programs developed for the earlier architectures. Memory is divided into segments of 64 KB. The maximum size of the linear address space in real-address mode is $2^{20}$ bytes.

The Pentium II has three separate processing modes: protected, real-address and system management. In protected mode, the processor can use any of the preceding memory models, flat, segmented or real. In real-address mode the processor can only support the real-addressing mode. In ths system management mode the processor uses an address space from the system management RAM (SMRAM). The memory model used in the SMRAM is similar to the real-addressing mode.

The Pentium II provides 16 registers for system processing and application programming. The first eight registers are the general-purpose registers and are used for logical and arithmetic operands, address calculations and for memory pointers. The registers are shown in Fig. 7.16. The EAX register is used as an accumulator for operands and the results of calculations. EBX is used as a pointer into the DS segment memory, ECX for loop and string operations, EDX for an I/O pointer, ESI for a pointer to data in the segment pointed to by the DS register; source pointer for string operations; EDI for a pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.

ESP is used for a stack pointer (in the SS segment) and the EBP register is used as a pointer to data on the stack (in the SS segment). The lower 16 bits of the registers correspond to those in the 8086 architecture and 80286 and can be referenced by programs written for those processors.

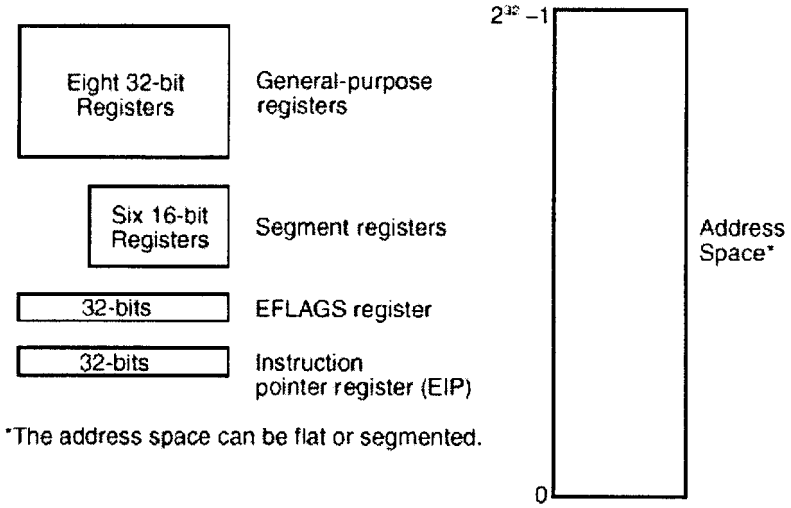



**Figure 7.16** Pentium® Pro processor basic execution environment

Six Segment registers hold a 16-bit address for segment addresses. The CS register handles code segment selectors where instructions are loaded from memory for execution. The CS register is used together with the EIP register that contains the linear address of the next instruction to be executed. The DS, ES, FS, and GS registers are data segment selectors that point to four separate data segments that the application program can access. The SS register is the stack segment selector for all stack operations. The processor contains a 32-bit EFLAGS register that various processor status, control and system flags that may be set by the programmer using special-purpose instructions.

The instruction pointer (EIP) register contains the address in the current code segment for the next instruction to be executed. It is incremented to the next instruction or it is altered to move ahead or backwards by a number of instructions when executing jumps, call interrupts and return instructions. The EIP register cannot be accessed directly by programs; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions. The EIP register controls program flow and is compatible with all Intel microprocessors, regardless of pre-fetching operations.

There are many different data types that the Pentium II processor supports. The basic data types are of 8-bit bytes, 2 byte or 16-bit words, double words of 4 bytes (32 bits) and quad words that are 8 bytes or 64 bits

in length. Additional data types are provided for direct manipulation of signed and unsigned integers, binary coded decimals (BCD), pointers, bit fields strings, floating point data types (to be used with the floating point unit (FPU)) and special 64-bit MMX data types.

A number of operands (zero or more) are allowed by the Intel architecture. The addressing modes fall into four categories: immediate, register, memory pointer and I/O pointer.

If the data are included as part of the instruction, then the addressing mode is immediate. Immediate operands are allowed for all arithmetic instructions except division, and must be smaller than the maximum value of an unsigned double word, $2^{32}$.

In register addressing mode, the source and destination operands can be given as any one of the 32-bit general-purpose registers (and their 16-bit subsets), the 8-bit general-purpose registers, the EFLAGS register and some system registers. The system registers are usually manipulated by implied means from the system instruction.

Source and destination operands in memory locations are given as a combination of the segment selector and the address offset within that segment. Most applications of this process are to load the segment register with the proper segment and then include that register as part of the instruction. The offset value is the effective address of the memory location that is to be referenced. This value may be direct or combinations of a base address, displacement, index and scale into a memory segment.

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register.

## Instruction Set

The Intel architecture can be classified in the family of complex instruction set computer (CISC) machines. The Pentium II has many types of powerful instructions for the system and application programmer. These instructions are divided into three groups: integer instructions (to include MMX), floating-point instructions and system instructions.

Intel processors include integer instructions for integer arithmetic, program control and logic functions. The subcategories of these types of instructions are data transfer, binary arithmetic, decimal arithmetic, logic, shift and rotate, bit and byte, control transfer, string, flag control, segment register and miscellaneous. The following paragraphs will cover the mostly used instructions.

The Move instructions allow the processor to move data to and from registers and memory locations provided as operands to the instruction. Conditional moves based on the value, comparison or of status bits are provided. Exchange, compare, push and pop stack operations, port transfers and data type conversions are included in this class.

The binary arithmetic instruction class includes integer add, add with carry, subtract, subtract with borrow, signed and unsigned multiply, signed and unsigned divide as well as instructions to increment, decrement, negate and compare integers are provided decimal arithmetic instructions. This class of instructions deals with the manipulation of decimal and ASCII values and adjusting the values after an add, subtract, multiply or divide functions. Logical AND, OR, XOR (Exclusive or) and NOT instructions are available for integer values. Shift arithmetic left and right include logical shift and rotate left and right, with and without carry manipulate integer operands of single and double word length.

The Pentium allows the testing and setting of bits in registers and other operands. The bit and byte functions are as follows: bit test, bit test and set, bit test and reset, bit test and complement, bit scan forward, bit scan reverse, set byte if equal/set byte if zero, and set byte if not equal/set byte if not zero.

The control instructions allow the programmer to define jump, loop, call, return, and interrupt, check for out-of-range conditions and enter and leave high-level procedure calls. The jump functions include instructions for testing zero, not zero, carry, parity, less or greater than conditions. Loop instructions use the ECX register to test conditions for loop constructs.

The Pentium II string instructions allow the programmer to manipulate strings by providing move, compare, scan, load, store, repeat, input and output for string operands. A separate instruction is included for byte, word and double word data types.

The MMX$^{TM}$ instructions execute on packed-byte, packed-word, packed-double-word, and quad word operands. MMX instructions are divided into subgroups of data transfer, conversion, packed arithmetic, comparison, logical shift and rotate, and state management. The data transfer instructions are MOVD and MOVQ for movement of double and quad words. Figure 7.17 shows the mapping from the special MMX registers to the mantissa of the floating-point unit (FPU). The Pentium II correlates the MMX registers and the FPU registers so that either the MMX instructions or the FPU instructions can manipulate values. Conversion instructions deal mainly with packing and unpacking of word and double word operands. The MMX arithmetic instructions add, subtract and multiply packed bytes, words and double words.

The floating-point instructions are those that are executed by the processor's floating-point unit (FPU). These instructions operate on floating-

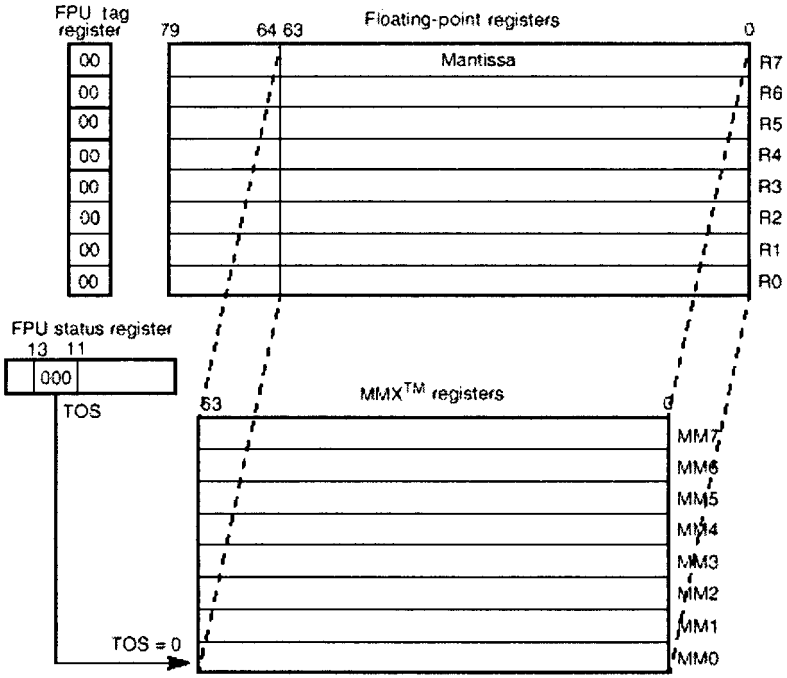



**Figure 7.17** Mapping of MMX registers to FPU

point (real), extended integer, and binary-coded decimal (BCD) operands. Floating-point instructions are divided into categories similar to the arithmetic instructions: data transfer, basic arithmetic, comparison, transcendental (trigonometry functions), load constants and FPU control.

System instructions are used to control the processor to support for operating systems and executives. These instructions include loading and storing of the global descriptor table (GDT) and the local descriptor table (LDT), task, machine status, cache, table look-aside buffer manipulation, bus lock, halt and for providing read performance information.

The EFLAGS instructions allow the state of selected flags in the EFLAGS register to be read or modified. Figure 7.18 shows the EFLAGS register.

## Hardware Architecture

The Pentium II is manufactured using Intel's 0.25 $\mu$m manufacturing process and contains over 7.5 million transistors. This process enables the

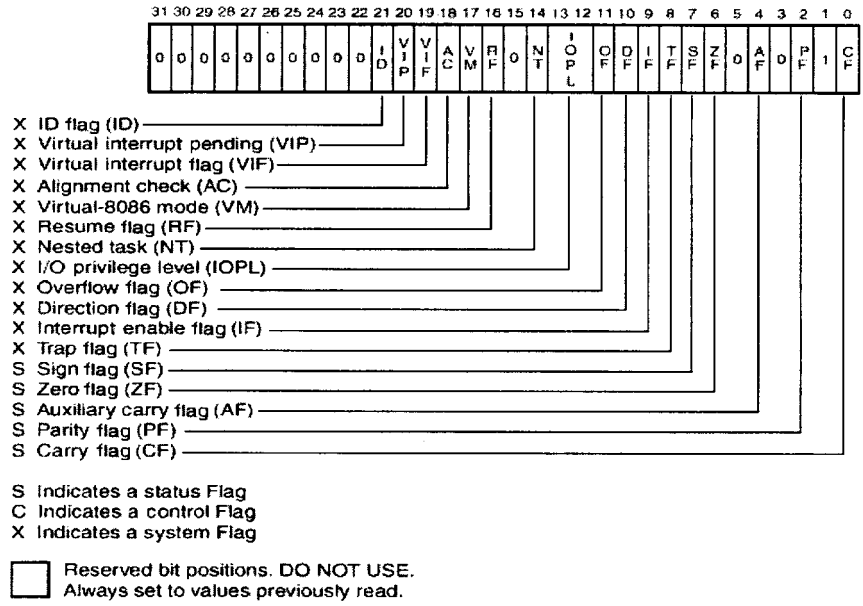

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID flag (ID)
X Virtual interrupt pending (VIP)
X Virtual interrupt flag (VIF)
X Alignment check (AC)
X Virtual-8086 mode (VM)
X Resume flag (RF)
X Nested task (NT)
X I/O privilege level (IOPL)
X Overflow flag (OF)
X Direction flag (DF)
X Interrupt enable flag (IF)
X Trap flag (TF)
S Sign flag (SF)
S Zero flag (ZF)
S Auxiliary carry flag (AF)
S Parity flag (PF)
S Carry flag (CF)
S Indicates a status Flag
C Indicates a control Flag
X Indicates a system Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

**Figure 7.18** EFLAGS register

Pentium II to run at clock speeds from 233 to 450 MHz. It contains a 32 KB L1 cache that is separated into a 16 KB instruction cache and a 16 KB data cache, and a 512 KB L2 cache that operates on a dedicated 64-bit cache bus. It supports memory cacheability for up to 4 GB of addressable memory space. It uses a dual independent bus (DIB) architecture for increased bandwidth and performance. The system bus speed is increased from 66 MHz to 100 MHz. It contains the MMX media enhancement technology for improved multimedia performance. It uses a pipelined floating-point unit (FPU) that supports IEEE standard 754, 32-bit and 64-bit formats and an 80-bit format. Packaged in the new single edge contact (SEC) cartridge to allow higher frequencies and more handling protection.

The Pentium II is a three-way superscalar, meaning it can fetch, decode, and execute up to three instructions per clock cycle, architecture. Program execution in the Pentium II is carried out by a twelve stage (fine grain) pipeline consisting of the following stages: instruction prefetch, length decode, instruction decode, rename/resource allocation, $\mu$OP scheduling/dispatch, execution, write back, and retirement. The pipeline can be broken down into three major stages with each of the twelve stages either supporting or contributing to these three stages. The three stages are: fetch/decode,

dispatch/execute, and retire. Figure 7.19 shows each of these stages, their interface through the instruction pool, connections to the L1 cache, and the bus interface.

The fetch/decode unit fetches the instructions in their original program order from the instruction cache. It then decodes these instructions into a series of micro-operations ($\mu$op) that represent the dataflow of that instruction. It also performs a speculative pre-fetch by also fetching the next instruction after the current one from the instruction cache. Figure 7.20 shows the fetch/decode unit.

The L1 instruction cache is a local instruction cache. The Next_IP unit provides the L1 instruction cache index, based on inputs from the branch target buffer (BTB). The L1 instruction cache fetches the cache line corresponding to the index from the Next_IP, and the next line (prefetch), and passes them to the decoder. The decoder is actually made up of three parallel decoders that accept this stream from the instruction cache. The three decoders are two simple instruction decoders and one complex instruction decoder. The decoder converts the instructions into triadic $\mu$ops (two logical sources, one logical destination per $\mu$op). Most instructions are converted directly into single $\mu$ops, some instructions are decoded into one-to-four $\mu$ops and the complex instructions require microcode which is stored in the microcode instruction sequencer. This microcode is just a set of preprogrammed sequences of normal $\mu$ops. After decoding the $\mu$ops are sent to the

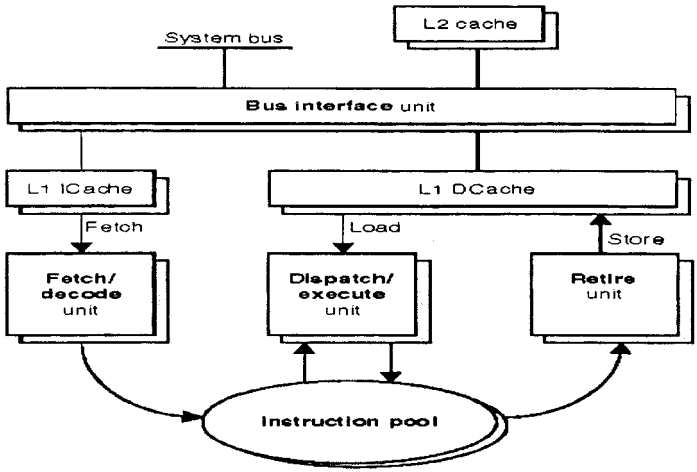



**Figure 7.19** Instruction execution stages (from Intel's *P6 Family of Processors Hardware Developer's Manual*)

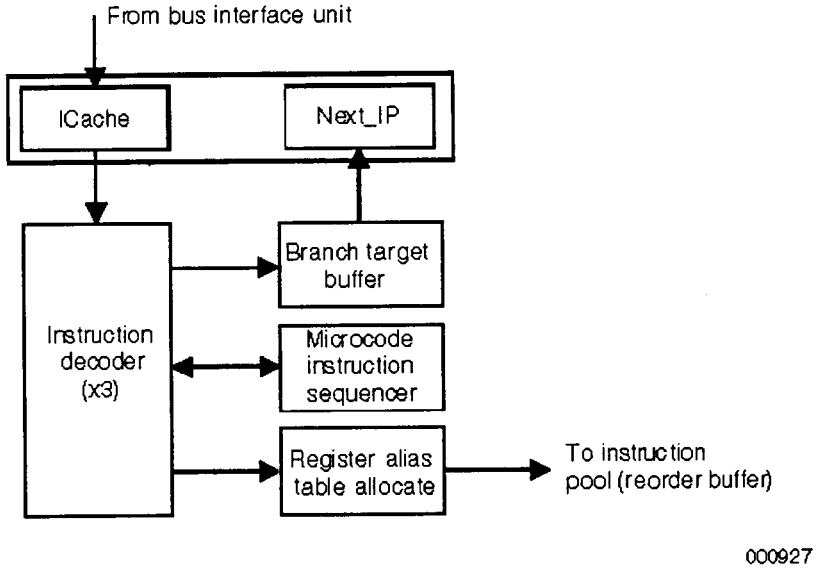



**Figure 7.20** Fetch/decode unit (from Intel's *P6 Family of Processors Hardware Developer's Manual*)

register alias table (RAT) unit which adds status information to the $\mu$ops and enters them into the instruction pool. The instruction pool is implemented as an array of content addressable memory called the reorder buffer (ROB).

Through the use of the instruction pool the dispatch/execute unit can perform instruction out of their original program order. The $\mu$ops are executed based on their data dependencies and resource availability. The results are then stored back into the instruction pool to await retirement based on program flow. This is considered speculative execution since the results may or may not be used based on the flow of the program. This is done in order to keep the processor as busy as possible at all times. The processor can schedule at most five $\mu$ops per clock cycle (3 $\mu$ops is typical), one for each resource port. Figure 7.21 shows the dispatch/execute unit.

The dispatch unit selects $\mu$ops from the instruction pool depending upon their status. If the status indicates that a $\mu$op has all of its operands then the dispatch unit checks to see if the execution resource needed by that $\mu$op is also available. If both are true, the reservation station removes that $\mu$op and sends it to the resource where it is executed. The results of the $\mu$op are later returned to the pool. If there are more $\mu$ops available than can be executed then the $\mu$ops are executed in a first-in/first-out (FIFO) order. The

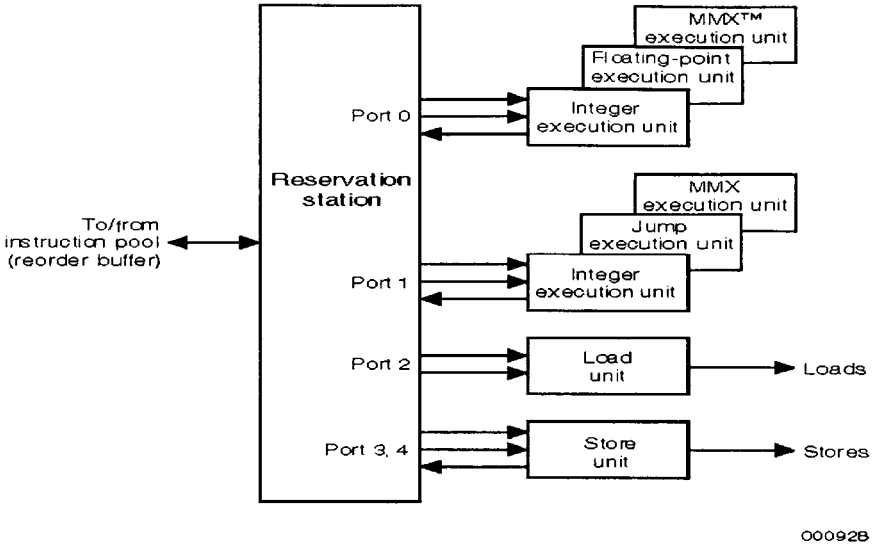



**Figure 7.21** Dispatch/execute unit (from Intel's *P6 Family of Processors Hardware Developer's Manual*)

core is always looking ahead for other instructions that could be speculatively executed, and is typically looking 20–30 instructions in front of the instruction pointer.

The retire unit is responsible for ensuring that the instructions are completed in their original program order. Completed means that the temporary results of the displatch/execute stage are permanently committed to memory. The combination of the retire unit and the instruction pool allows instructions to be started in any order but always be completed in the original program order. Figure 7.22 shows the retire unit.

Every clock cycle, the retire unit checks the status of $\mu$ops in the instruction pool. It is looking for $\mu$ops that have executed can be removed from the pool. Once removed, the original target of the $\mu$ops is written based on the original instruction. The retire unit must not only notice which $\mu$ops are complete, it must also re-impose the original program order on them. After determining which $\mu$ops can be retired the retire unit writes the results of this cycle's retirements to the retirement register file (RRF). The retire unit is capable of retiring 3 $\mu$ops per clock.

As shown previously the instruction pool removes the constraint of linear instruction sequencing between the traditional fetch and execute phases.

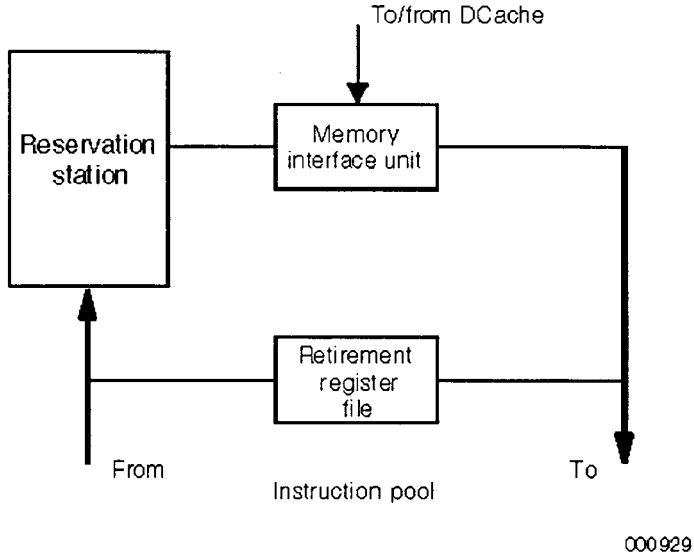



**Figure 7.22** Retire unit (from Intel's *P6 Family of Processors Hardware Developer's Manual*)

The bus interface unit is responsible for connecting the three internal units (fetch/decode, dispatch/execute, and retire) to the rest of the system. The bus interface communicates directly with the L2 cache bus and the system bus. Figure 7.23 shows the bus interface unit.

The memory order buffer (MOB) allows loads to pass other loads and stores by acting like a reservation station and re-order buffer. It holds suspended loads and stores and re-dispatches them when a blocking condi-

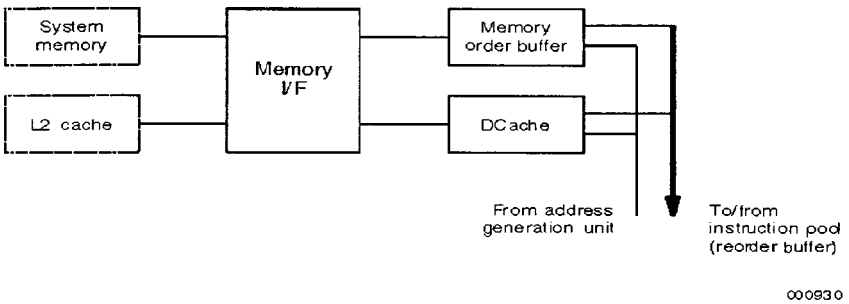



**Figure 7.23** Bus interface unit (from Intel's *P6 Family of Processors Hardware Developer's Manual*).

tion (dependency or resource) disappears. Loads are encoded into a single $\mu$op since they only need to specify the memory address to be accessed, the width of the data being retrieved, and the destination register. Stores need to provide a memory address, a data width, and the data to be written. Stores therefore require two $\mu$ops, one to generate the address and one to generate the data. These $\mu$ops must later re-combine for the store to complete. Stores are also never re-ordered among themselves. A store is dispatched only when both the address and the data are available and there are no older stores awaiting dispatch.

A combination of three processing techniques enables the processor to be more efficient by manipulating data rather than processing instructions sequentially. The three techniques are multiple branch prediction, data flow analysis, and speculative execution.

Multiple branch prediction uses algorithms to predict the flow of the program through several branches. While the processor is fetching instructions, it's also looking at instructions further ahead in the program.

Data flow analysis is the process performed by the dispatch/execute unit. By analyzing data dependencies between instructions it schedules instructions to be executed in an optimal sequence, independent of the original program order.

Speculative execution is the process of looking ahead of the program counter and executing instructions that are likely to be needed. The results are stored in a special register and only used if needed by the actual program path. This enables the processor to stay busy at all times, and if the results are needed increases performance.

The bus structure of the Pentium II is referred to as a dual independent bus (DIB) architecture. The DIB is used to aid processor bus bandwidth. By having two independent buses the processor can access data from either bus simultaneously and in parallel. The two buses are the L2 cache bus and the system bus.

The cache bus refers to the interface between the processor and the L2 cache, which for the Pentium II is mounted on the substrate with the core. The L2 cache bus is 64 bits wide and runs at half the processor core frequency.

The system bus refers to the interface between the processor, system core logic and other bus agents. The system bus does not connect to the cache bus. The system bus is also 64 bits wide and runs at about 66 MHz, and is pipelined to allow simultaneous transactions. Figure 7.24 shows a block diagram of the Pentium II processor.

The Pentium II contains two integer and two floating-point units that can all operate in parallel, with all sharing the same instruction decoder, sequencer, and system bus.

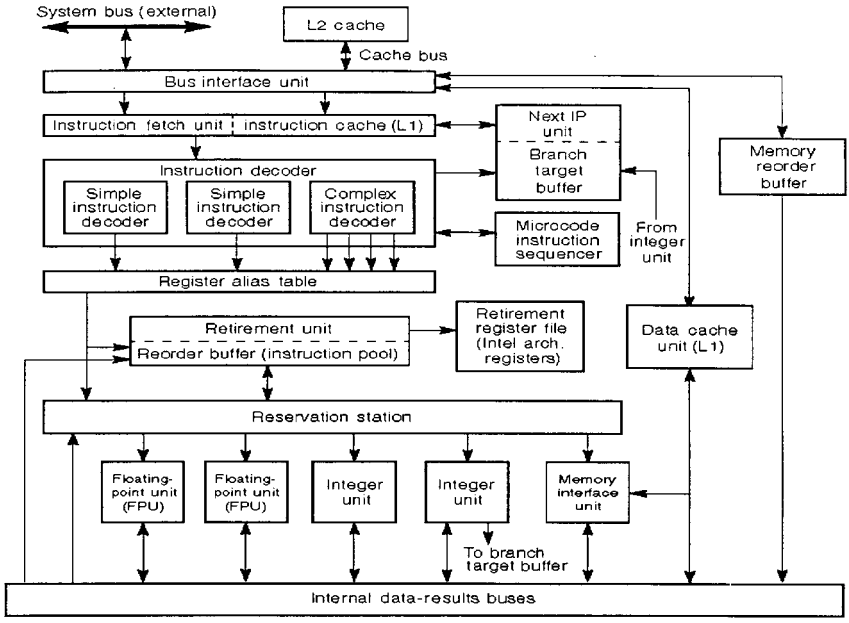



**Figure 7.24** Bus structure (from Intel's *The Intel Architecture Software Developer's Manual)*

The FPU supports real, integer, and BCD-integer data types and the floating-point processing algorithms defined in the IEEE 754 and 854 Standards for floating-point arithmetic. The FPU uses eight 80-bit data registers for storage of values. Figure 7.25 shows the relationship between the integer and floating-point units.

The Pentium II uses input/output ports (I/O ports) to transfer data. I/O ports are created in system hardware by circuitry that decodes the

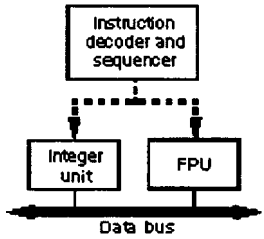



**Figure 7.25** Integer and floating-point units (from Intel's *The Intel Architecture Software Developer's Manual*)

control, data, and address pins on the processor. The I/O port can be an input port, an output port, or a bidirectional port. The Pentium II allows I/O ports to be accessed in either of two ways: as a separate I/O address space, or memory-mapped I/O. I/O addressing is handled through the processor's address lines. The Pentium II uses a special memory-I/O transaction on the system bus to indicate whether the address lines are being driven with a memory address or an I/O address.

Accessing I/O ports through the I/O address space is handled through a set of I/O instructions and a special I/O protection mechanism. This guarantees that writes to I/O ports will be completed before the next instruction in the instruction stream is executed. Accessing I/O ports through memory-mapped I/O is handled with the processor's general-purpose move and string instructions, with protection provided through segmentation or paging.

The Pentium II has two levels of cache, the L1 cache and the L2 cache. Memory is cacheable for up to 4 GB of addressable memory space.

The L1 cache is 32 KB, that is divided into two 16 KB units to form the instruction cache and the data cache. The instruction cache is four-way set associative and the data cache is two-way set associative each with a 32-byte cache line size. The L1 cache operates at the same frequency as the processor and provides the fastest access to the most frequently used information. If there is a L1 cache miss then the L2 cache is searched for the data.

The Pentium II supports between 256 KB and 1 MB of L2 cache with 521 KB most common. The L2 cache is four-way set associative with a 32-byte cache line size. The L2 cache uses a dedicated 64-bit bus to transfer data between the processor and the cache. Cache coherency is maintained through the MESI (modified, exclusive, shared, invalid) snooping protocol. The L2 cache can support up to four concurrent cache accesses as long as they are to different banks. Figure 7.26 shows the cache architecture of the Pentium II.

### MMX Technology

Considered the most significant enhancement to the Intel architecture in the last 10 years. This technology provides the improved video compression/decompression, image manipulation, encryption and I/O processing needed for today's multimedia applications. These improvements are achieved through the use of the following: single instruction, multiple data (SIMD) technique, 57 new instructions, eight 64-bit wide MMX registers, and four new data types.

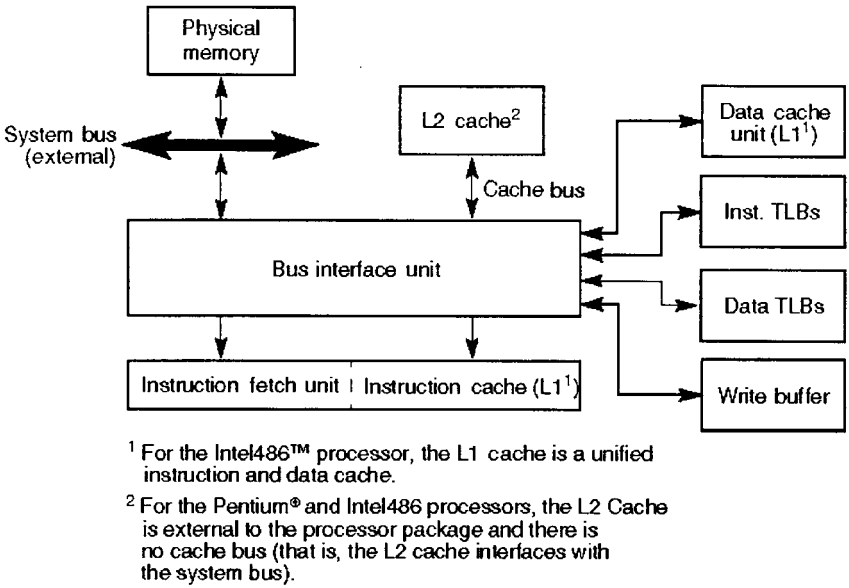

Physical memory
System bus (external)
L2 cache2
Cache bus
Data cache unit (L1[1])
Inst. TLBs
Bus interface unit
Data TLBs
Instruction fetch unit
Instruction cache (L1[1])
Write buffer
[1] For the Intel486™ processor, the L1 cache is a unified instruction and data cache.
[2] For the Pentium® and Intel486 processors, the L2 Cache is external to the processor package and there is no cache bus (that is, the L2 cache interfaces with the system bus).

**Figure 7.26** Cache architecture (from Intel's *The Intel Architecture Software Developer's Manual*)

This technique allows for the parallel processing of multiple data elements by a single instruction. This is accomplished by performing an MMX instruction on one of the MMX packed data types. For example an add instruction performed on two packed bytes would add eight different values with the single add instruction.

The 57 new instructions cover the following areas: basic arithmetic, comparison operations, conversion instructions, logical operations, shift operations, and data transfer instructions. These are general-purpose instructions that fit easily into the parallel pipelines of the processor and are designed to support the MMX packed data types.

MMX contains eight 64-bit registers (MM0–MM7) that are accessed directly by the MMX instructions. These registers are only used to perform calculations on the MMX data types. The registers have two data access modes: 64-bit and 32-bit access mode. The 64-bit mode is used for transfers between MMX registers and the 32-bit mode is for transfers between integer registers and MMX registers.

The four new data types included in MMX are packed byte, packed word, packed doubleword, and quadword (see Fig. 7.27). Each of these data types are a grouping of signed and unsigned fixed-point integers, bytes,

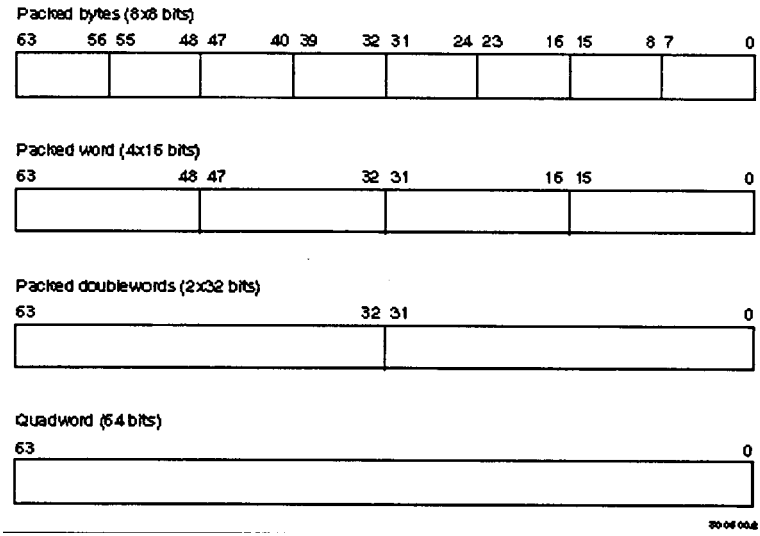

Packed bytes (8x8 bits)
63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0
Packed word (4x16 bits)
63 48 47 32 31 16 15 0
Packed doublewords (2x32 bits)
63 32 31 0
Quadword (64 bits)
63 0

**Figure 7.27** MMX data types (from Intel's *The Intel Architecture Software Developer's Manual*)

words, doublewords and quadwords into a single 64-bit quantity. These are then stored in the 64-bit MMX registers. Then the MMX instruction executes on all of the values in the register at once.

The single edge connect cartridge (SEC) is a metal and plastic cartridge that completely encloses the processor core and the L2 cache. To enable high-frquency operations, the core and L2 cache are surface mounted directly to a substrate inside the cartridge. The cartridge then connects to the motherboard via a single edge connector. The SEC allows the use of high performance BSRAMs (widely available and cheaper) for the L2 cache. The SEC also provides better handling protection for the processor.

### 7.6.7 Compaq Computer Corporation's SP700 Workstation

This section is extracted from a set of white papers (listed in the References section) from Compaq Computer Corporation. Compaq workstations are classified in three categories: affordable performance (AP), scalable performance (SP) and extreme performane (XP). The AP series includes the AP200, the AP400, and the AP500. They are based on Intel Pentium II and Pentium III processors operating at up to 500 MHz. They offer 2D and entry- and mid-range 3D graphics capability. The SP700 uses Intel

Pentium II Xeon and Pentium III Xeon processors running at up to 550 MHz, and offers entry, mid-range and enhanced 3D graphics. The XP1000 uses an Alpha 21264 processor running at 500 MHz. It offers entry, mid-range and enhanced 3D graphics capability. Both Windows NT and UNIX operating systems are used. We will focus on the SP700 in this section. Refer to manufacturer's literature for details on the other systems.

The SP700 uses a highly parallel system architecture utilizing Intel Pentium II Xeon and Pentium III Xeon processors and Windows NT operating system. The target applications for the system are: computer-aided design (CAD) and computer-aided manufacturing (CAM), computer-aided engineering (CAE), digital content creators (DCC), electronic analysis and risk management and geographic information systems (GIS).

The SP700 utilizes multiple data paths that offer a high degree of parallelism for data delivery to key subsystems, such as memory and I/O. Memory and I/O requests are processed in parallel, reducing system bottlenecks and therefore delivering higher application performance. Utilization of wide, high-speed data pipelines increases the throughput to critical subsystems, such as processor, memory and cache, resulting in higher application performance. Critical system resources reside on separate buses to help balance throughput, improve system efficiency and increase performance. Figure 7.28 shows the structure of the SP700.

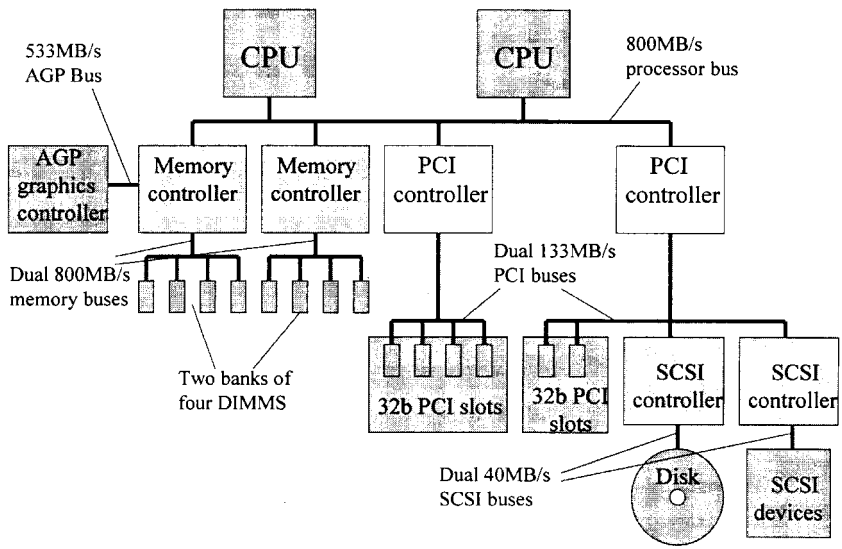



**Figure 7.28** System architecture of the SP700

The Intel Pentium II Xeon processor used in the SP700 has the following features: it is designed specially for workstation and server environments; it uses a high-performance balanced architecture for processor, memory, graphics and I/O; it offers built-in multiprocessing capabilities. It offers a dynamic execution micro-architecture with dual independent buses, multiprocessing support for up to eight processors with built-in cache coherency, full-speed level 2 cache (512K~2M) and error checking and correction (ECC) memory.

The SP700 memory subsystem (Fig. 7.29) includes two parts: memory controller and the memory. There are two memory controllers that can process memory requests in parallel, thus increasing the overall memory bandwidth. The memory controllers of the SP700 have the following features:

- Process memory requests in parallel.
- One memory controller associates with four dual in-line memory module (DIMM) slots.
- Total memory bandwidth is 1.6 GB/s.
- Support up to 4 GB memory.

Compared with the single in-line memory module (SIMM), the DIMM has the following two advantages:

- Higher bus widths: DIMM supports 64 or 72 bits bus width, while SIMM supports only 32 or 36 bits bus width.

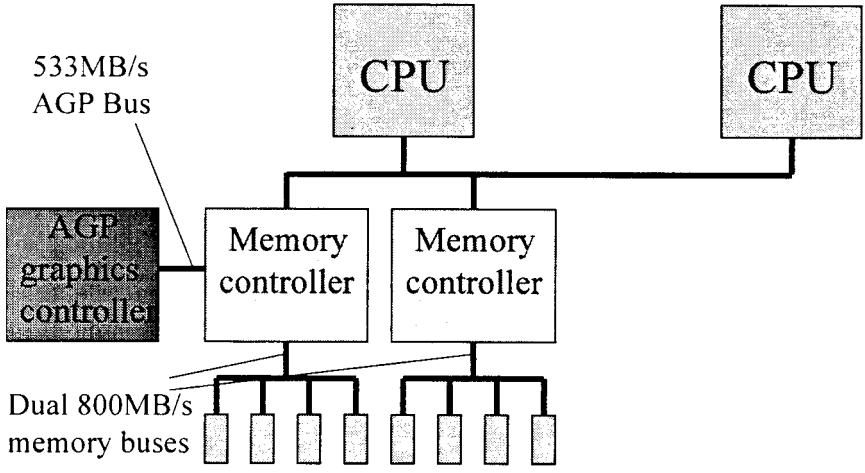



**Figure 7.29** SP700 memory subsystem

- Larger memory capacities: Given the same number of memory slots, the maximum memory of a system using DIMMs is more than one using SIMMs.

In addition, the Synchronous DRAM (SDRAM) also has several advantages over other types of memory. Compared with EDO DRAM, the SDRAM provides:

- Increased performance: SDRAM can support Pentium II Xeon 450 MHz processor, whereas EDO DRAM cannot
- Faster bus speed: SDRAM can run up to 100 MHz, while the maximum bus speed that EDO DRAM can run is 66 MHz.

SDRAM is better suited to handle graphics and other complex applications than EDO DRAM because of its higher bus speed.

Some workstation applications require large I/O bandwidth. For example, the video editing involves significant disk and I/O bandwidth. Such applications can take full advantage of the SP700 architecture, since it significantly reduces these bottlenecks by incorporating enhanced subsystem resources to accommodate the increased data traffic from the multiple processors.

The standard peripheral component interconnect (PCI) bus is 32 bits wide and operates at 33 MHz. Each PCI provides an I/O bandwidth of up to 133 MB/s, which is usually shared by many key peripherals, such as small computer system interface (SCSI) controllers, redundant array of independent disk (RAID) controllers, and network interface controllers (NIC). PCI features include:

- Burst mode – transfers data in bursts. After the initial address is provided, multiple data sets are transmitted at once.
- Bus mastering – supports full bus mastering, which allows peripheral controllers to access main memory independent of the processor.
- High-bandwidth options – expandable to 64-bit and 66 MHz.

As shown in Fig. 7.30, the SP700 uses two independently operating PCI buses (that is, dual-peer PCI buses) to increase system I/O bandwidth. Because each PCI bus runs independently, it is possible to have two PCI bus masters transferring data simultaneously. With dual-peer PCI buses, each bus can provide peak bandwidth in parallel with the other controller, allowing an aggregate I/O bandwidth of 267 MB/s.

The architecture also includes an I/O cache between the PCI bus and the processor bus that improves system concurrency, reduces latency for many PCI bus master accesses to system memory, and makes more efficient

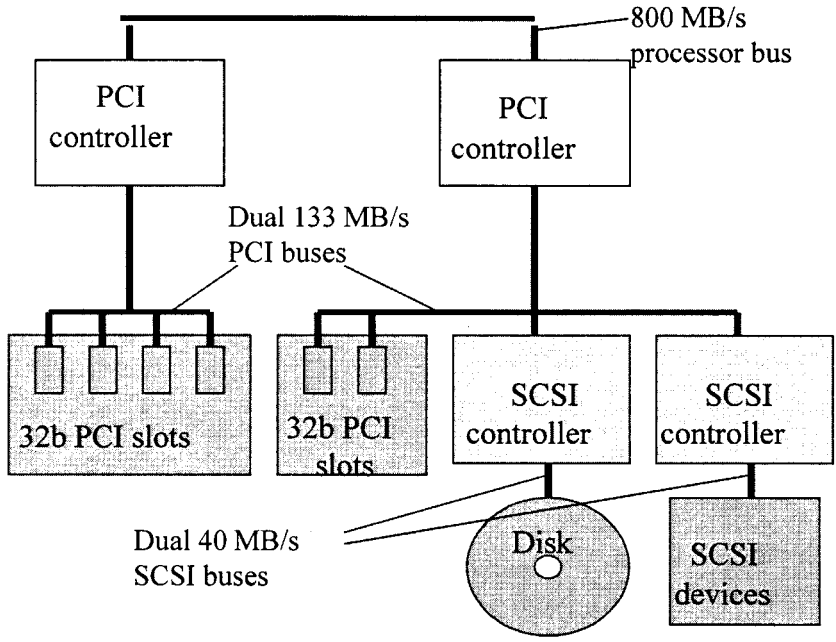



**Figure 7.30** The SP700 I/O structure

use of the processor bus. Because of the PCI bus bandwidth of only 133MB/s each, bottlenecks sometimes occur. The processor with an 800 MB/s bandwidth has to wait for the PCI to transfer the data. The I/O cache is a temporary buffer between the PCI bus and the processor bus. When a PCI bus master requests data accesses to system memory, the I/O cache controller automatically reads a full cache line (32 bytes) from the system memory at the processor transfer rate (800 MB/s) and stores it in the I/O cache. If the PCI bus master is reading memory sequentially (which is typical), subsequent read requests can be serviced from the I/O cache. Likewise, when a PCI master writes data, the data is stored in the I/O cache until the cache contains a full line. It may lead to an accompanying problem of cache coherent transaction. When a device on the PCI bus writes to main memory, the microprocessor must perform a snoop operation to make sure the data does not exist in the microprocessor cache memory.

Also, the dual PCI buses allow key peripherals to be connected to separate buses to balance system resources. By adding peripherals on separate PCI buses, devices do not have to compete for access on the same PCI bus, resulting in better performance and increased system throughput.

Finally, dual PCI buses also allow for greater system I/O integration and expandability by supporting up to 12 PCI devices. This allows the SP700 to provide six available PCI expansion slots while also integrating other PCI components, such as SCSI and network controllers, on the system board. The PCI bridge approach of other workstations does extend the PCI bus and allows more devices to be connected; however, greater traffic is caused on the single PCI bus and performance is degraded.

The SP700 uses dual, independent channel wide-ultra SCSI controllers which utilize a 16-bit bus with a maximum transfer rate of 40 MB/s. The dual wide-ultra SCSI implementation doubles the bandwidth to the disk subsystem, providing an aggregate bandwidth of 80 MB/s when compared to the single SCSI bus systems. Dual bus architecture allows balancing of the storage subsystem work load and performance by dividing high-performance peripherals on separate buses. In addition, performance can be optimized by separating high-performance peripherals such as RAID arrays from slower, non wide-ultra SCSI devices, such as tape backup devices on their own SCSI controller.

Other key I/O technologies employed by the SP700 workstation include:

- Two IEEE 1394 connections (requires Windows 98 or Windows NT 5.0). IEEE 1394 connection features are: high speed of 100/200/400 MB/s; real-time data transmission bandwidth for time-sensitive applications, such as audio and video.
- RAIDs.
- Two Universal Serial Bus (USB) ports that enable hot plug-and-play of 127 computer peripherals. Automatically configured USB peripherals include telephones, modems, CD-ROM drivers, and printers. USB supports transfer rates up to 12 Mbits/s, compared to 115.2 KB/s for serial ports and 2 Mb/s for enhanced parallel ports. The improved rate will accommodate a new generation of peripherals, including MPEG-2 video-based products and digitizers.
- NIC-Faster Ethernet (10/100 Mb/s) with 6 KB buffer memory lowers CPU utilization at high throughput. Other features include full duplex, autosensing, and autonegotiating.

Today's graphics and video demands are increasing and beginning to push the limits of the PCI bus. The accelerated graphics port (AGP) is a new industry-standard bus interface specially designed for 3D graphics applications to alleviate the strain on the PCI bus. The SP700 includes an AGP graphics controller. As shown in Fig. 7.31, AGP provides a dedicated bus connection between the graphics controller, processor bus, and system main memory. It provides higher-quality, faster graphics performance for 3D

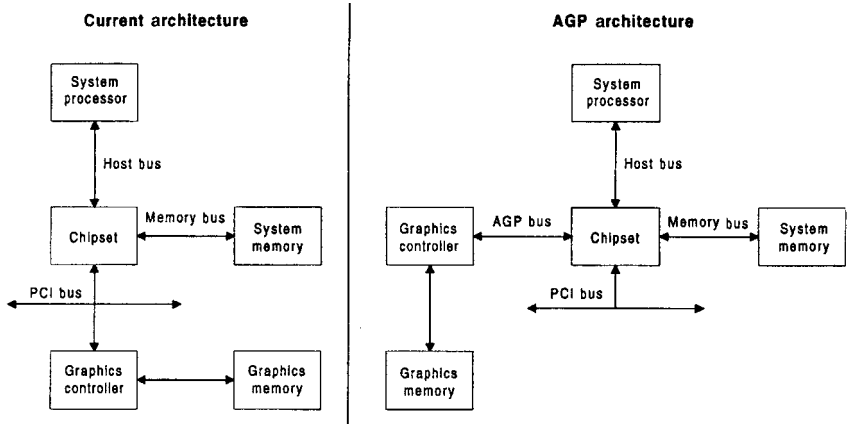



**Figure 7.31** Current architecture vs. AGP architecture

rendering using large-size texture mapping and video applications such as conferencing or DVD playback requesting steady image stream from system memory to the graphics frame buffer for display. With AGP, some large 3D rendering data structures persistent with an application, such as texture data, are shifted from local graphics memory into the less expensive system memory. The AGP bus provides a dedicated pipeline from the graphics controller to the system memory. The bandwidth load and memory size can be balanced between system memory and local graphics buffer. This lowers the system cost and frees the PCI bus for other performance-critical peripherals. On the SP700, the AGP 2× port is directly connected to one of the memory controllers. The fastest performance is achieved when memory is accessed from the primary controller.

AGP is based on a set of enhancements to PCI in the following aspects.

- It has higher bus speed, increased bandwidth with 1×, 2× and 4× modes. AGP 2× uses "double pumping" as shown in Fig. 7.32 that allows the graphics controller to transfer data on both the falling and rising clock edges. AGP 2× runs at 32 bits and 66 MHz with a maximum data transfer rate of 533 MB/s which is four times faster than PCI.
- On the PCI bus, data requests are processed sequentially; that is, the first request's address and data phases must complete before the next one begins. The AGP chipset has request buffers that allows the graphics controller to queue up to 256 read or write requests.

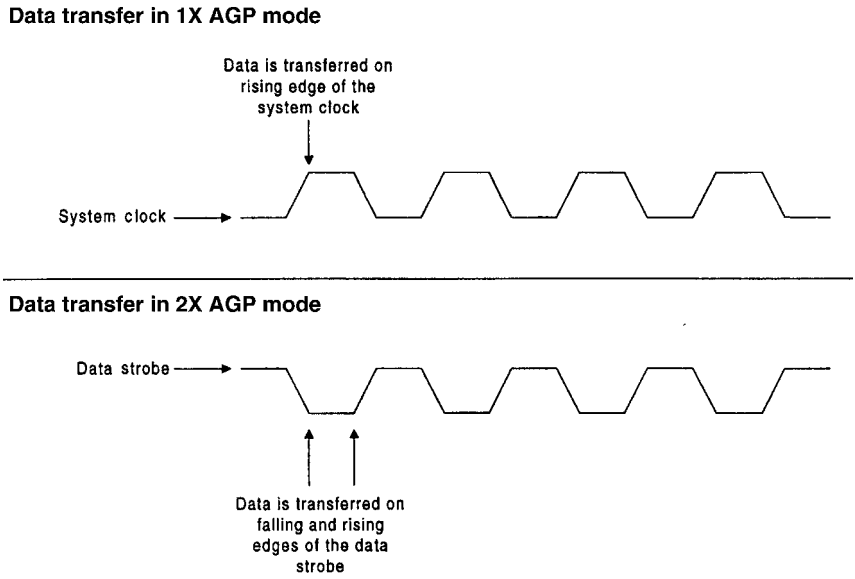

Data transfer in 1X AGP mode
Data is transferred on rising edge of the system clock
System clock
Data transfer in 2X AGP mode
Data strobe
Data is transferred on falling and rising edges of the data strobe

**Figure 7.32** Data transfers in 1× and 2× AGP modes

Request buffers also separate (demultiplex) the address and data phases. This enables request pipelining. Therefore multiple addresses can be submitted simultaneously which provides a much higher sustained bandwidth than the serial transfer allowed by PCI. Further, AGP requests are submitted on the sideband addressing (SBA) bus, freezing the address/data bus for data transfers as shown in Fig. 7.33.

There are two AGP usage models for 3D rendering: direct memory access (DMA) and execute. In the DMA model, the local graphics memory is treated as primary graphics memory. 3D structures are stored in system memory but are not executed directly from this memory. When needed, the 3D structures are moved to primary graphics memory.

With the execute model, both the system memory and the local graphics memory are treated as primary graphics memory. Data structures allocated to system memory are executed directly by the rendering engine. The graphics controller directly references system memory; therefore, a contiguous view of system memory is essential. However, data is not written to system memory in contiguous blocks. An address translation

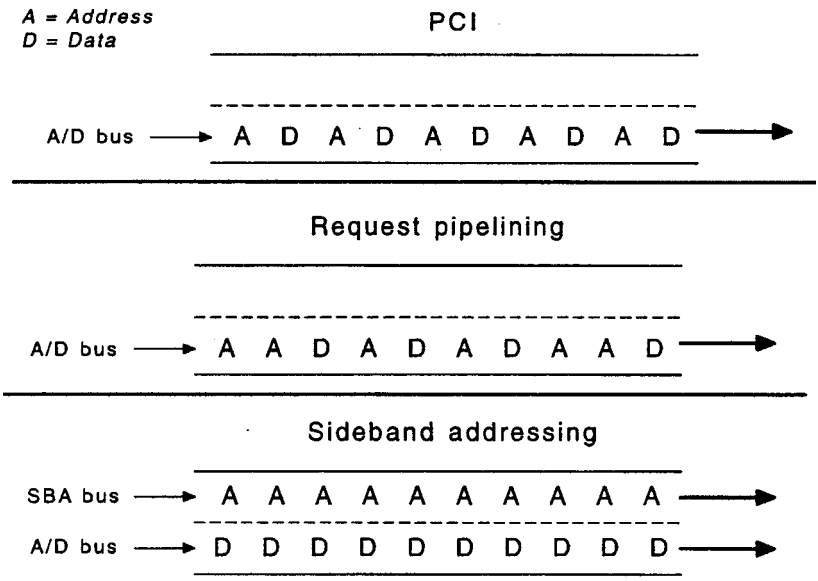

A = Address
D = Data
PCI
A/D bus
A D A D A D A D A D
Request pipelining
A/D bus
A A D A D A D A A D
Sideband addressing
SBA bus
A A A A A A A A A A
A/D bus
D D D D D D D D D D

**Figure 7.33** Address and data transfer with PCI, pipelining, and sideband addressing

mechanism is required to remap the "virtual", or AGP device address space, to the actual 4 KB-page physical address space (Fig. 7.34). Compaq uses a graphics address remapping table (GART), which is built into the AGU chip set, to perform the remapping. In addition, a two-level look-up table is used and allows a large GART to be located on smaller, non-contiguous memory blocks. Read merging allows the AGP 2 × system to combine requests to consecutive memory locations into a single cache line, thus improving graphics performance by reducing the number of reads from memory.

### 7.6.8 Alternative Workstation Architectures

Intel's GX Architecture (Fig. 7.35) is targeted for the workstation market. It is a successor to the Intel LX architecture introduced in 1997, with the addition of enhanced AGP graphics, a 100 MHz Advanced Gunning Transistor Logic (AGTL)+ system bus, and supports for 100 MHz ECC SDRAM memory. The GX architecture uses up to two Intel Pentium II or Pentium II Xeon processors. A single memory controller with maximum bandwidth of 800 MB/s is used. In this architecture, two processors must compete with each other for access to subsystems because of the limited overall system band-

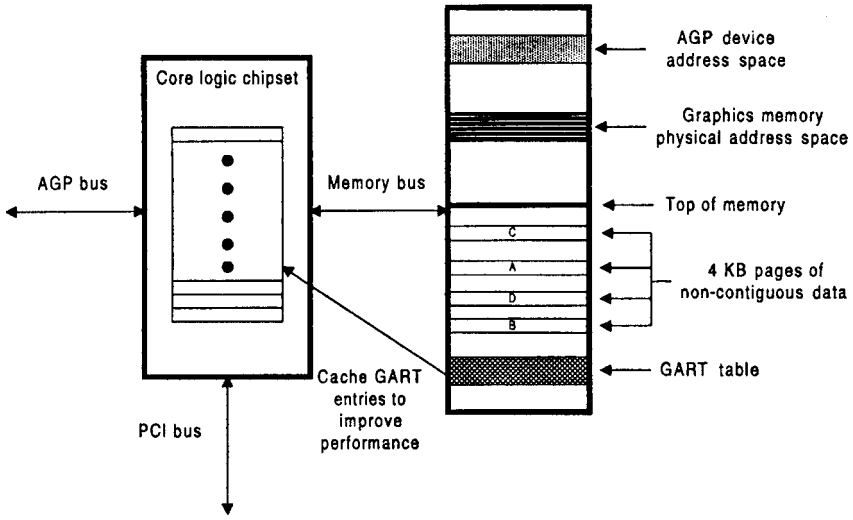



**Figure 7.34** GART table entry caching

width. Scalability is degraded by limiting the processors to four concurrent transactions. In contrast, the dual memory controllers in the SP700 provide more flexibility and better expandability.

The SP700 is designed to deliver maximum bandwidth to critical subsystems by exploiting concurrency whenever possible. Compared with GX architecture, the SP700 offers superior performance, scalability, and expandability.

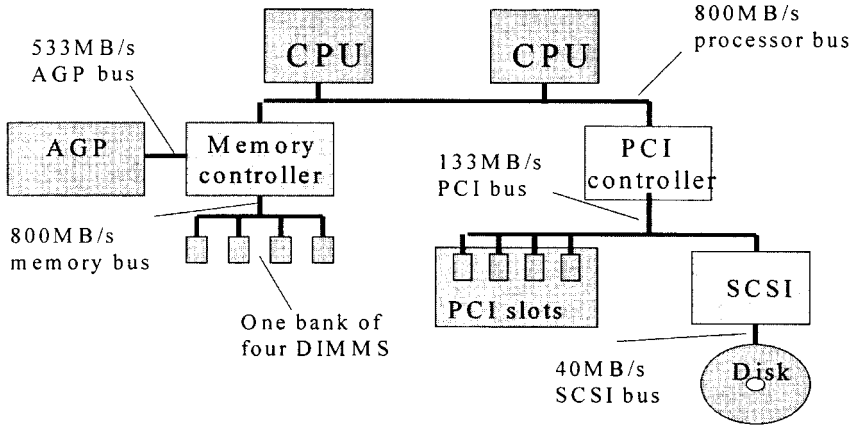



**Figure 7.35** Intel's GX architecture

### A Crossbar Switch Architecture by Sun Microsystems – Ultra Port Architecture (UPA)

The UPA provides multiple, independent paths to system memory as shown in Fig. 7.36. It has processor, memory, PCI, and graphics ports. By allowing separate paths to system memory, the crossbar switch improves performance of both I/O traffic and processor cycles. A crossbar switch for a single processor bus, memory bus, and I/O bus can be implemented cost effectively. However, a crossbar switch is an expensive solution with today's silicon technology in a system with several buses since all the buses must go into a single chip that has sufficient pins for each bus. The UPA avoids this problem by forcing the PCI buses and graphics bus to all share the same bus into the crossbar switch. AGP is frequently the highest bandwidth peripheral and is better served with a direct path to main memory as implemented by Compaq. Compaq chose not to use a crossbar switch in any other subsystem because a better architectural solution was possible. The peak UPA memory bandwidth is 1.9 GB/s. A single memory controller using older EDO memory supports only one open memory page at a time.

### Unified Memory Architecture (UMA) by Silicon Graphics, Inc.

Figure 7.37 depicts SGI's UMA architecture. The processor and graphics controller share one memory pool that is connected by a single bus with a peak bandwidth of up to 2.1 GB/s. The unified memory is cost effective.

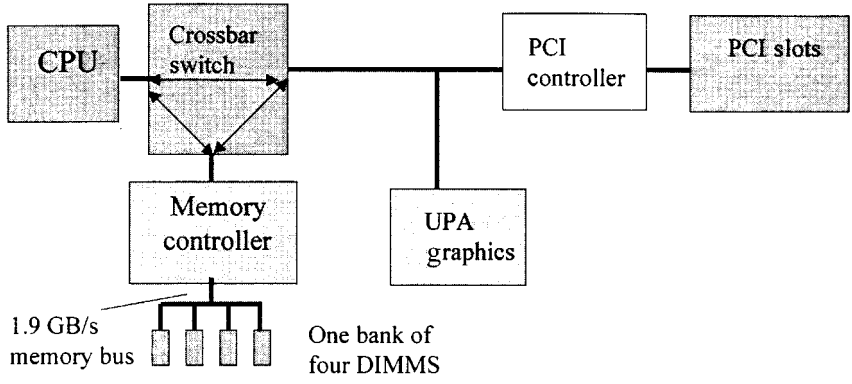



**Figure 7.36** The ultra port architecture

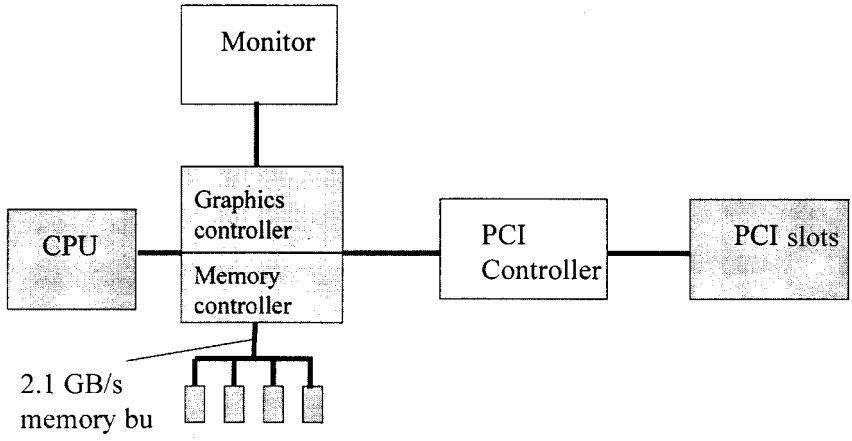



**Figure 7.37** SGI's UMA architecture

However, the refreshing monitor consumes 334 MB/s at 85 Hz at a screen resolution of 1280 × 1024. The actual bandwidth available to processor and graphics controller for other tasks is much reduced.

## 7.7 SUMMARY

Several enhancements to the processor structure of ASC have been described in this chapter, through the details of the processor structures of commercially available machines. An attempt was made to trace the evolution of commercial architectures through these examples. In addition to the processor structure, the structure of the system of which the processor is a component also influences the performance. Representative system structure examples were provided in this chapter. A more detailed analysis of the evolution would be interesting and informative. The reader is referred to the manufacturer's manuals for further details on these processors and systems. As can be seen by these examples and the others in subsequent chapters of this book, it is not just the advances in hardware technology alone that influence the architecture. As the hardware technology permits, more and more features oriented toward aiding easier system- and application-software development become the architecture primitives. Chapters 8 through 11 provide further examples.

## REFERENCES

*Accelerated Graphics Port Technology*, ECG Technology Communications, ECG081/0898, Houston, TX: Compaq Computer Corp., August, 1998.

Baron, R. J. and Higbie, L. *Computer Architecture*, Reading, MA: Addison Wesley, 1992.

*Compaq Professional Workstation SP700 Key Technologies* White Paper, ECG 078/0898, Houston, TX: Compaq Computer Corp., September, 1998.

Cramer, W. and Kane, G., *68000 Microprocessor Handbook*, Berkeley, CA: Osborne McGraw-Hill, 1986.

*Highly Parallel System Architecture of the Compaq Professional Workstation SP700*, ISSD Technology Communications, ECG067/1198, Houston, TX: Compaq Computer Corp., November, 1998.

Hill, M. D., Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.

*IBM System 37O Principles of Operation*, GA22-7000. Poughkeepsie, NY.: IBM Corporation, 1976.

*Intel 8080 Microcomputer Systems Users Manual*, Santa Clara, CA: Intel Corp., 1977.

*Intel Architecture Software Developer's Manuals – Volume 1: Basic Architecture (24319001.pdf); Volume 2: Instruction Set Reference (24319101.pdf) and Volume 3: System Programming Guide (24319201.pdf)* at http://www.intel.-com/design/PentiumII/manuals, Santa Clara, CA: Intel Corp.

M6502 Hardware Manual, Norristown, PA: MOS Technology Inc., 1976.

*MC68000 Technical Features*. Phoenix, Ariz. Motorola Semiconductor Products.

*MC68000 Users Manual*, Austin, TX: Motorola, 1982.

*MCS-80 Users Manual*, Santa Clara, CA: Intel, 1977.

Mano, M. M. *Computer Systems Architecture*, Englewood Cliffs, NJ: Prentice-Hall, 1992.

*P6 Family of Processors Hardware Developer's Manual*, Santa Clara, CA: Intel Corp., http://www.intel.com/design/PentiumII/manuals/24400101.pdf

Patterson, D. A. and Hennessey, J. L. *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann, 1990.

*PDP-11 Processor Handbook*, Maynard, MA: Digital Equipment Corp., 1978.

*Pentium II Processor Technology*, ECG Technology Communications, ECG046/0897, Houston, TX: Compaq Computer Corp., August, 1997.

*Pentium II Xeon Processor Technology Brief*, Order Number: 243792-002, Santa Clara, CA: Intel Corp., 1998.

*Second-Generation Highly Parallel System Architecture vs. the Intel 440 BX/GX AGP set in the Workstation Market*, ISSD Technology Communications, ECG092/1198, Houston, TX: Compaq Computer Corp., November, 1998.

Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.

Tannenbaum, A. S. and Goodman, J. R. *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall, 1998.

*VAX Hardware Handbook*. Maynard, Mass.: Digital Equipment Corp., 1979.

## PROBLEMS

7.1 Study the processor and system characteristics of a large-scale computer, a minicomputer, and a microprocessor system you have access to, with reference to the features described in this chapter.

7.2 Assume that the processor status register of an 8-bit machine contains the following flags: CARRY, ZERO, OVERFLOW, EVEN PARITY, and NEGATIVE. Determine the value of each of these flags after an ADD operation on two operands A and B for the various values of A and B shown below:

| A | B |
|---|---|
| +1 | –1 |
| 12 | –6 |
| 31 | 31 |
| 12 | –13 |
| 127 | –127 |

7.3 Represent the following numbers in IBM and IEEE standard floating-point notation:

$46.24 \times 10^{-3}$ $2.46 \times 10^{2}$

7.4 Design a 4-bit-per-level, eight-level stack using shift registers. In addition to allowing PUSH and POP operations, the stack must generate an "overflow" signal when an attempt to PUSH into a full stack is made and an "underflow" signal when an empty stack is popped.

7.5 Design the stack of Problem 7.4 using a RAM.

7.6 Assume that ASC has a stack pointer register (SP) that is initiated to 0 when power is turned on. Assume also two instructions LSP and SSP for loading and storing the SP from and into a memory location, respectively. The following operations are required:

a. PUSH: TL⟵ACC.
b. POP: ACC⟵TL.
c. ADD: SL⟵TL + SL; POP

Write subroutines for these operations using the ASC instruction set including the two new instructions LSP and SSP.

7.7 Write the ASC microinstruction sequences for PUSH, POP, and ADD instruction in Problem 7.6.

7.8 In a machine such as IBM 370 that does not allow indirect addressing mode, how can an operand whose address is in a memory location be accessed?

7.9 Assume that a branch instruction with a PC-relative mode of addressing is loated at X1. If the branch is made to location X2, what is the value of the address field of the instruction? If the address field is 10 bits long, what is the range of branches possible?

7.10 Using the notation of Chapter 4, express the effective address of the operand for each of the addressing modes described in this chapter.

7.11 Write programs for zero-, one-, two- and three-address machines to evaluate $ax^2 + bx + c$, given the values for $x, a, b$, and $c$.

7.12 Given that

F = M + N + P – (Q – R – S)/T + V).

where F, M,...., V each represent a memory location, generate code for

a. A stack machine.
b. A one-address machine.
c. A two-address machine.
d. Assume 8 bits of opcode and 16 bits for an address and compute the length of the program in each of the above cases.
e. How many memory cycles are needed to execute the above programs?
f. Assume that the stack is ten levels deep. What is the effect on the program in (a) if the stack is restricted to four levels?

7.13 Answer Problem 7.12(e) assuming that the memory of each of the machines is organized as eight bits/word and only one memory word can be accessed per memory cycle.

7.14 An example of a macroinstruction is a TRANSLATE instruction. The instruction replaces an operand with a translated version of itself, the translated version being derived from a table stored in a specified memory location. Assume that in ASC the operand is located in the accumulator and the beginning address of the table is located in the second word of the instruction (i.e., the instruction is two words long). The accumulator content thus provides the offset into the table. Write a microprogram for TRANSLATE.

7.15. Assuming that a machine using base-displacement addressing mode has $N$-bit base registers, investigate how the compilers for that machine access data and instructions in blocks that are larger than $2^N$ words.

7.16 A machine to compute the roots of the quadratic: $ax^2 + bx + c = 0$ is needed. Develop the instruction set needed for zero-, one-, two-, and three-address architectures.

7.17 The instruction set of a machine has the following number and types of instructions:

10 3-address instructions
36 2-address instructions (including 10 memory reference instructions)
50 1-address instructions
30 0-address instructions.

The machine has eight 16-bit registers. All the instructions are 16-bit register-to-register, except for the memory reference, load, store and branch instructions. Develop an instruction format and an appropriate opcode structure.

7.18 Develop the subroutine call and return instructions for all the three types of machines. Assume that a stack is available.

7.19 Discuss the relative merits and effects on the instruction cycle implementation of the following parameter-passing techniques:
a. Pass the parameters in registers.
b. Pass the parameters on stack.
c. Pass the parameter addresses in registers.
d. Pass the parameter addresses on stack.

7.20 In a four-address machine, the fourth address in the instruction corresponds to the address of the next instruction to be fetched. This means that the PC is not needed. What types of instructions would not utilize the fourth address?

# 8
# Memory System Enhancement

In Chapter 3 we introduced the *memory system hierarchy* common to most modern-day computer systems. This hierarchy consists of the following levels:

1. CPU registers
2. Cache memory, a small, fast, random-access memory block
3. Primary (main) memory, the random-access memory from which the processor accesses all programs and data (via the cache memory)
4. Secondary (mass) memory consisting of semi-random access and sequential access memory elements such as magnetic disks and tapes.

The fastest memory is at level 1, and the speed decreases as we move towards higher levels. The cost per bit is highest at level 1 and lowest at level 4. The major aim of the hierarchical memory design is to enable a speed–cost tradeoff that will provide a memory system of the desired capacity with the highest speed possible at the lowest cost.

As seen by the example systems described in earlier chapters, additional levels have been added to the above hierarchy in recent computer systems. In particular, it is common to see two levels of cache (between the processor and the main memory), typically called level-1 and level-2 or on- and off-chip cache, depending on the system structure. In addition, a disk cache could be present (between the primary and secondary memory), to enable faster disk access.

Let us consider the cost of the memory system with an n-level hierarchy. Let $C_i$ ($i = 1$ through $n$) be the cost per bit of the $i$th level in the above hierarchy, while $S_i$ is the capacity (i.e., total number of bits) at the $i$th level. Then the average cost per bit ($C_a$) of the memory system is given by:

$$C_a = \frac{\sum_{i=1}^{n} C_i S_i}{\sum_{i=1}^{n} S_i} \tag{8.1}$$

Typically, we would like to have $C_a$ as close to $C_n$ as possible. Since $C_i >>> C_{i+1}$, in order to minimize the cost it requires that $S_i <<< S_{i+1}$.

This chapter describes the design of memory hierarchies as two broad types of enhancements to the single-level memory system discussed in previous chapters: *speed enhancement* techniques that include the cache memory and *size enhancement* techniques that include *virtual memory* schemes.

## 8.1 SPEED ENHANCEMENT

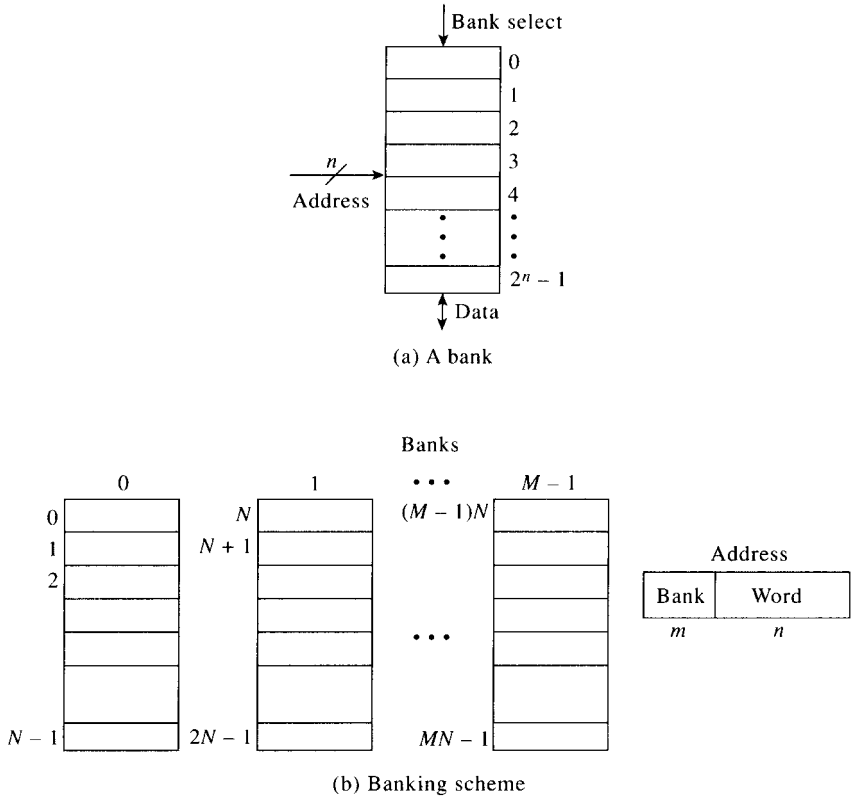
Traditionally, memory cycle times have been much longer than processor cycle times; this speed gap between the memory and the processor means that the processor must wait for memory to respond to an access request. With the advances in hardware technology, faster semiconductor memories are now available and have replaced core memories as primary memory devices. But the processor–memory speed gap still exists, since the processor hardware speeds have also increased. Several techniques have been used to reduce this speed gap and optimize the cost of the memory system.

The obvious method of increasing the speed of the memory system is by using a higher-speed memory technology. Once the technology is selected, the access speeds can be increased further by judicious use of address decoding and access techniques. Six such techniques are described in the following sections.

### 8.1.1 Banking

Typically, the main memory is built out of several physical-memory modules. Each module is a memory bank of a certain capacity and consists of a memory address register and a memory buffer register. In semiconductor memories, each module corresponds to either a memory IC or a memory board consisting of several memory ICs (as described in Chapter 3).

Figure 8.1 shows the memory banking addressing scheme. Here, the consecutive addresses lie in the same bank. If each bank contains $2^n = N$ words and if there are $2^m = M$ banks in the memory, then the system MAR would contain $n + m$ bits. Figure 8.1(a) shows the functional model of a bank. The *bank select* signal (BS) is equivalent to the *chip select* (CS) of Chapter 3. In Figure 8.1(b), the most significant $m$ bits of the MAR are




Note: $2^m = M$
$2^n = N$

**Figure 8.1** Memory banking

decoded to select one of the banks, and the least significant $n$ bits are used to select a word in the selected bank.

In this scheme, given that the data and program are stored in different banks, the next instruction can be fetched from the program bank, while the data required for the execution of the current instruction is being fetched from the data bank, thereby increasing the speed of memory access.

Since the subsequent addresses are in the same bank, during the sequential program execution process, accesses will have to be made from the same program bank. This scheme, however, limits the instruction fetch to one instruction per memory cycle. Its advantage is that even if one bank fails, the other banks provide continuous memory space, and the

operation of the machine is unaffected (except for reduced memory capacity).

### 8.1.2 Interleaving

Interleaving the memory banks is a technique to spread the subsequent addresses to separate physical banks in the memory system. This is done by using the low-order bits of the address to select the bank, as shown in Fig. 8.2.

The advantage of the interleaved memory organization is that the access request for the next word in the memory can be initiated while the current word is being accessed, in an overlapping manner. This mode of access increases the overall speed of memory access. The disadvantage of this scheme is that if one of the memory banks fails, the complete memory system becomes inoperative.

An alternative organization is to implement the memory as several subsystems, each subsystem consisting of several interleaved banks. Figure 8.3 shows such an organization.

There is no general guideline to select one addressing scheme over the other among the three described above. Each computer system uses its own scheme. The basic aim of these schemes is to spread subsequent memory references over several physical banks so that faster accessing is possible.

DEC PDP-11/45 interleaves memory blocks in pairs of 8K word banks. If the two banks are magnetic core memories, the memory cycle time averages 650 nanoseconds compared to the 850 ns individual cycle time.

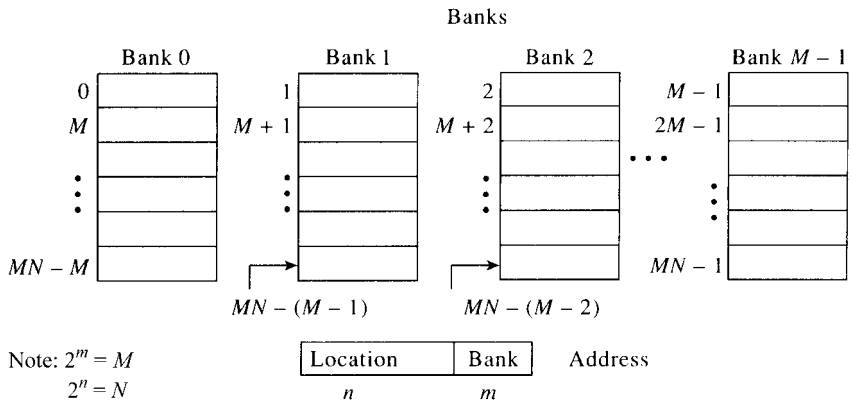



**Figure 8.2** Interleaving

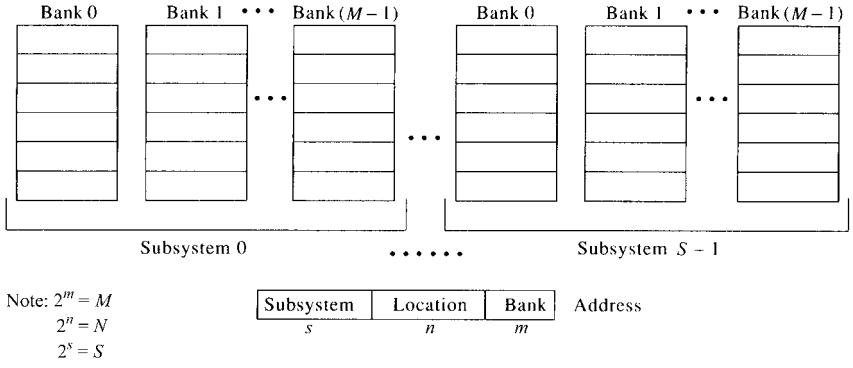



**Figure 8.3** Interleaved subsystems

Primary memory in Control Data Corporation's (CDC) 6600 is organized into 131,072 words, each 60 bits long in 32 banks of 4,096 words per bank. These banks are arranged in an interleaved fashion, which provides a high degree of random-access overlap and block transfer speed. Each word in the primary memory is identified by an 18-bit address. The least significant 5 bits of the address are used to select one of the 32 banks and the remaining bits to select a word within the bank. Consecutive addresses, therefore, lie in separate physical memory blocks.

### 8.1.3 Multiport Memories

Multiple-port (multiport) memories are available in which each port corresponds to an MAR and an MBR. Independent accesses to the memory can be made from each port. The memory system resolves the conflicts between the ports on a priority basis. Multiport memories are useful in an environment where more than one device accesses the memory. Examples of such systems are a single processor system with a direct-memory access (DMA), I/O controller (see Chapter 6), and a multiprocessor system with more than one CPU (see Chapter 11).

### 8.1.4 Wider Word Fetch

IBM 370 fetches 64 bits (two words) in each memory access. This enhances execution speed, because the second word fetched most likely contains the next instruction to be executed, thus saving a "wait" for the fetch. If the second word does not contain the required instruction (during a jump, for

example) a new fetch is required. This scheme was also used in the IBM 7094, a machine designed in the early sixties.

### 8.1.5 Instruction Buffer

Providing a first-in, first-out (FIFO) buffer (or queue) between the CPU and the primary memory enhances the instruction fetch speed. Instructions from the primary memory are fed into the buffer at one end and the CPU fetches the instructions from the other end, as shown in Fig. 8.4. As long as the instruction execution is sequential, the two operations of filling the buffer (prefetching) and fetching from the buffer into CPU can go on simultaneously. But when a jump (conditional or unconditional) instruction is executed, the next instruction to be executed may or may not be in the buffer. If the required instruction is not in the buffer, the fetch operation must be directed to the primary memory and the buffer is refilled from the new memory address. If the buffer is large enough, the complete range of the jump (or loop) may be accommodated in the buffer. In such cases, the CPU can signal the buffer to FREEZE, thereby stopping the prefetch operation. Once the loop or jump is satisfied, both FETCH operations can continue normally.

The buffer management requires hardware components to manage the queue (e.g., check for queue full or queue empty) and mechanisms to identify the address range in the buffer and to freeze and unfreeze the buffer.

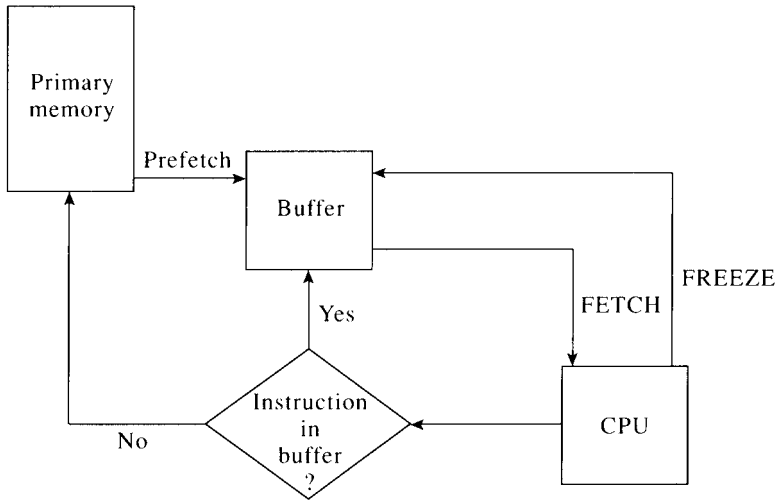



**Figure 8.4** Instruction buffer

CDC 6600 uses an instruction buffer that can store eight 60-bit words, that is, 16 to 32 instructions, since the instructions are either 15 or 30 bits long. Figure 8.5 shows the instruction buffer organization. Instructions from main memory are brought into the buffer through a buffer register. The lowest level of the buffer is transferred to the instruction register for execution purposes while the contents of the buffer move up one position. A new set of instructions (60 bits) enters the buffer through the buffer register. When a branch instruction is encountered, if the address of the branch is within the range of the buffer, the next instruction is retrieved from it. If not, instructions are fetched from the new memory address.

INTEL 8086 (refer to Fig. 7.13) processor uses a 6-byte long instruction buffer organized as a queue.

### 8.1.6 Cache Memory

Analyses of memory reference characteristics of programs have shown that typical programs spend most of their execution times in a few main modules

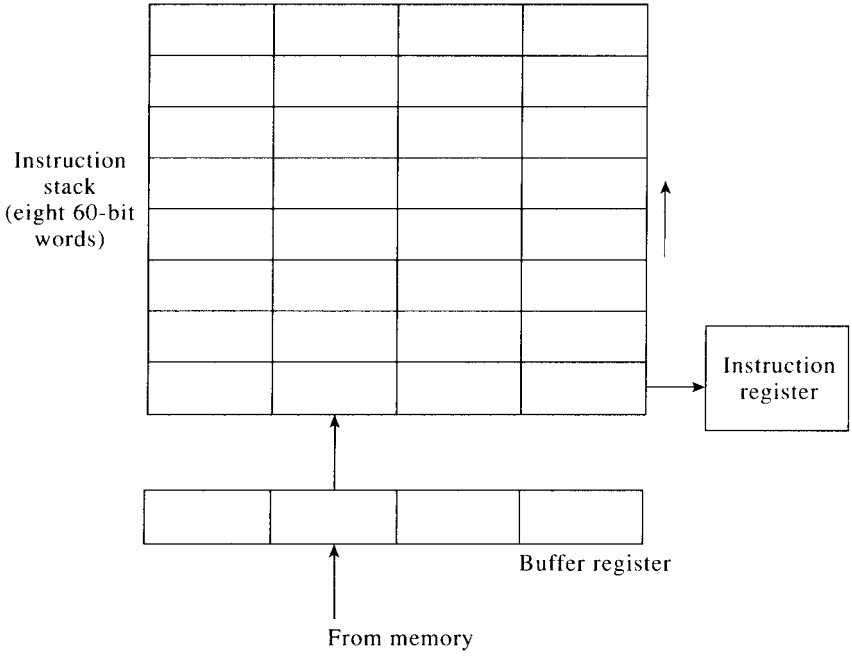



**Figure 8.5** CDC 6600 instruction buffer

(or routines) and tight loops. Therefore, the addresses generated by the processor (during instruction and data accessing) over short time periods tend to cluster around small regions in the main memory. This property, known as the *program locality principle* is utilized in the design of cache memory scheme. The cache memory is a small (of the order of $\frac{1}{2}$ K to 2K words) but fast (of the order of 5 to 10 times the speed of main memory) memory module inserted between the processor and the main memory. It contains the information most frequently needed by the processor, and the processor accesses the cache rather than the main memory, thereby increasing the access speed.

Figure 8.6 shows a cache memory organization. Here, the primary memory and cache are each divided into blocks of $2^n = N$ words. The block size depends on the reference characteristics implied by the program locality and the main memory organization. For instance, if the main memory is organized as an $N$-way interleaved memory, it is convenient to make each block $N$ words long, since all $N$ words can be retrieved from the main memory in one cycle. We will assume that the cache capacity is $2^b = B$ blocks. In the following discussion, we will refer to the block in the main memory as a main memory *frame* and a block in the cache memory as a *block* or a *cache line*.

In addition to the *data* area (with a total capacity of $B \times N$ words), the cache also has a *tag* area consisting of $B$ tags. Each tag in the tag area identifies the address range of the $N$ words in the corresponding block in the cache. If the primary memory address A contains $p$ bits, and the least significant $n$ bits are used to represent the $N$ words in each frame, the remaining $(p - n)$ bits of the address form the tag for that block, whereby the tag is the beginning address of the block of $N$ words.

When the processor references a primary memory address A, the cache mechanism compares the tag portion of A to each tag in the tag area. If there is a matching tag (i.e., a cache *hit*), the corresponding cache block is retrieved from the cache and the least significant $n$ bits of A are used to select the appropriate word from the block, to be transmitted to the processor. If there is no matching tag (i.e., a cache *miss*), first the frame corresponding to A is brought into the cache and then the required word is transferred to the processor.

---

**Example 8.1** Figure 8.7 shows the cache mechanism for a system with 64 KB of primary memory and a 1 KB cache. There are 8 bytes per block.

---

In general, a cache mechanism consists of the following three functions:

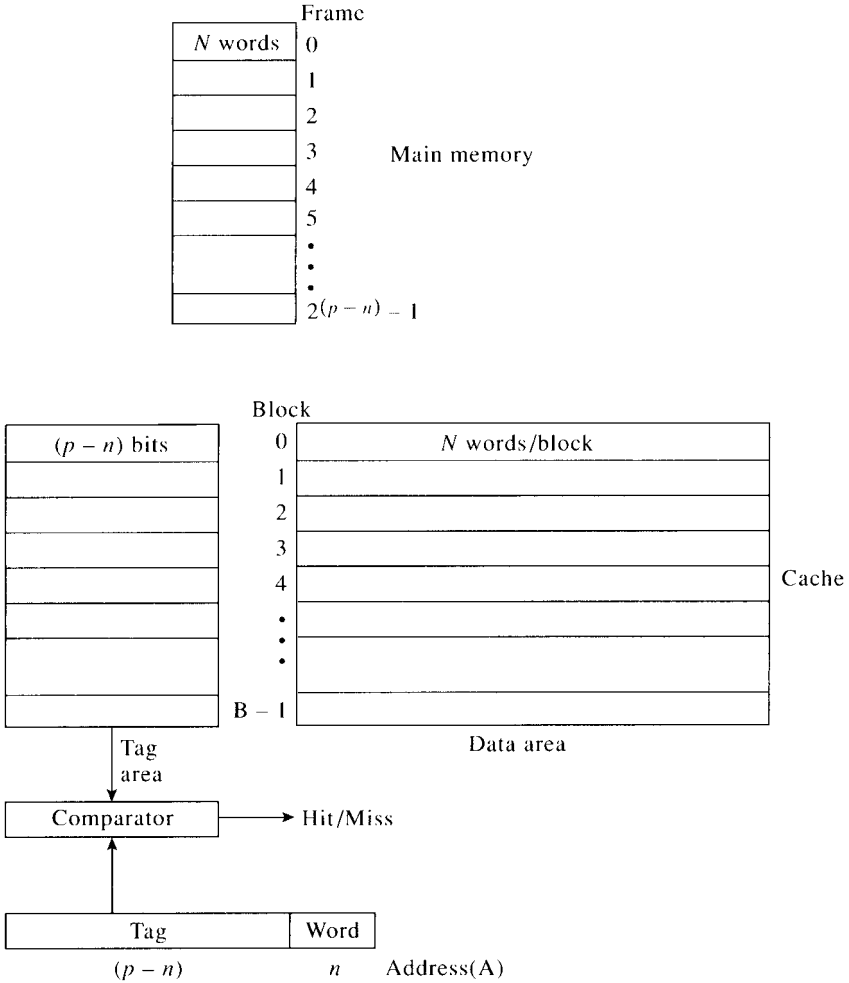



**Figure 8.6** Cache mechanism

1. *Address translation function*: determines if the referenced block is in the cache or not and handles the movement of blocks between the primary memory and the cache.
2. *Address mapping function*: determines where the blocks are to be located in the cache.
3. *Replacement algorithm*: determines which of the blocks in the cache can be replaced, when space is needed for new blocks during a cache miss. These functions are described next.

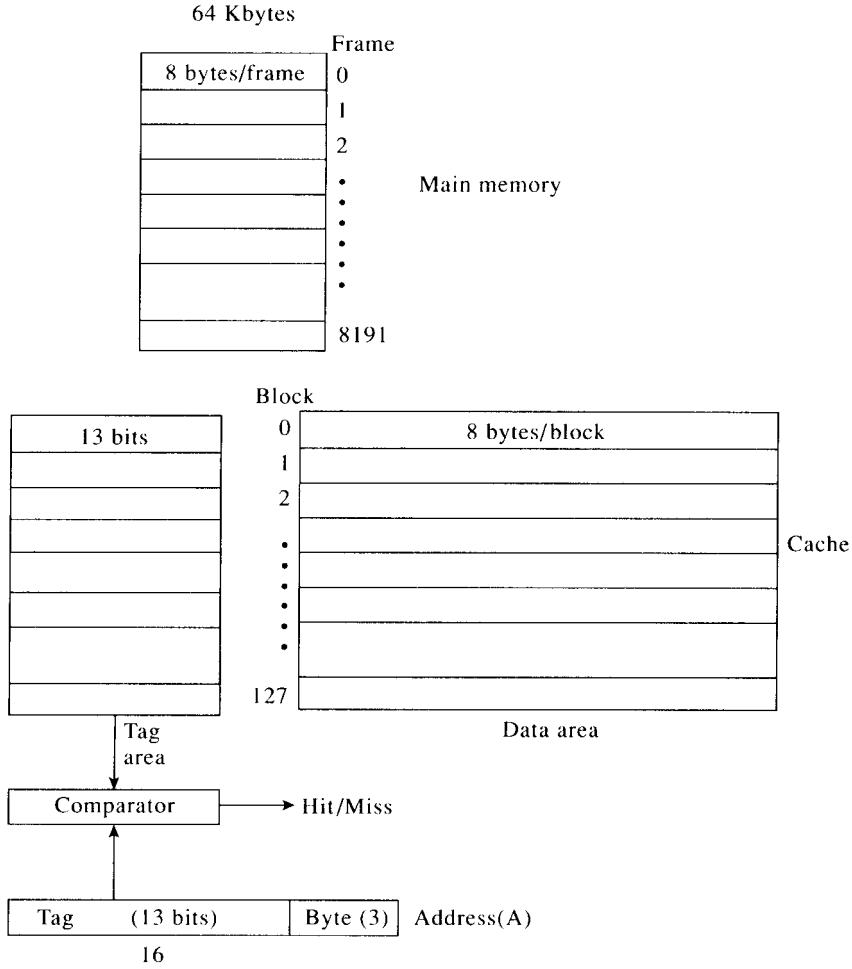



**Figure 8.7** An example of a cache mechanism (Example 8.1)

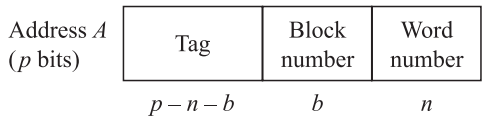
## Address Translation

The address translation function of Fig. 8.6 is simple to implement but enforces an $N$-word boundary to each frame. Thus, even if the referenced address A corresponds to the last word in the frame, all the words of the frame are transferred to the cache. If the translation function were to be general, in the sense that an $N$-word frame starting from an arbitrary address $A$ is transferred to the cache, then each tag would have to be $p$ bits long, rather than $(p - n)$ bits. In addition to this overhead, in this

general scheme it will not always be possible to retrieve $N$ words from $A$ in one memory cycle, even if the primary memory is $N$-way interleaved. Because of these disadvantages of the general addressing scheme, the scheme shown in Fig. 8.6 is the most popular one.

**Address Mapping Function.** In the address mapping scheme of Fig. 8.6, a main memory frame can occupy any of the cache blocks. Therefore, this scheme is called a *fully associative* mapping scheme. The disadvantage with this scheme is that all the $B$ tags must be searched in order to determine a hit or miss. If $B$ is large, this search can be time consuming. If the tag area is implemented using a RAM, an average of $B/2$ RAM cycles are needed to complete the search. An alternative is to implement the tag area of the cache using an associative memory. Then the search through $B$ tags requires just one associative memory (compare) cycle. (Refer to Chapter 3 for the description of associative memory.)

One way of increasing the tag search speed is to use the least significant $b$ bits of the tag to indicate the cache block in which a frame can reside. Then, the tags would be only $(p - n - b)$ bits long, as shown below:




This mapping is called *direct mapping* since each primary memory address $A$ maps to a unique cache block. The tag search is now very fast since it is reduced to just one comparison. But, this mechanism divides the $2^p$ addresses in the main memory into $B = 2^b$ partitions. Thus, $2^{p-b}$ addresses map to each cache block. Note that the addresses that map to the same cache block are B $\times$ $N$ words apart in the primary memory. If the consecutive references are to the addresses that map to the same block, the cache mechanism becomes inefficient since it requires a large number of block replacements.

---

**Example 8.2** Figure 8.8 shows the direct-mapping scheme for the memory system of Example 8.1. Note that now the data area of the cache contains only 32 blocks of 8 bytes each.

---

A compromise between the above two types of mapping is called the *set-associative* mapping, in which the B cache blocks are divided into $2^k = K$

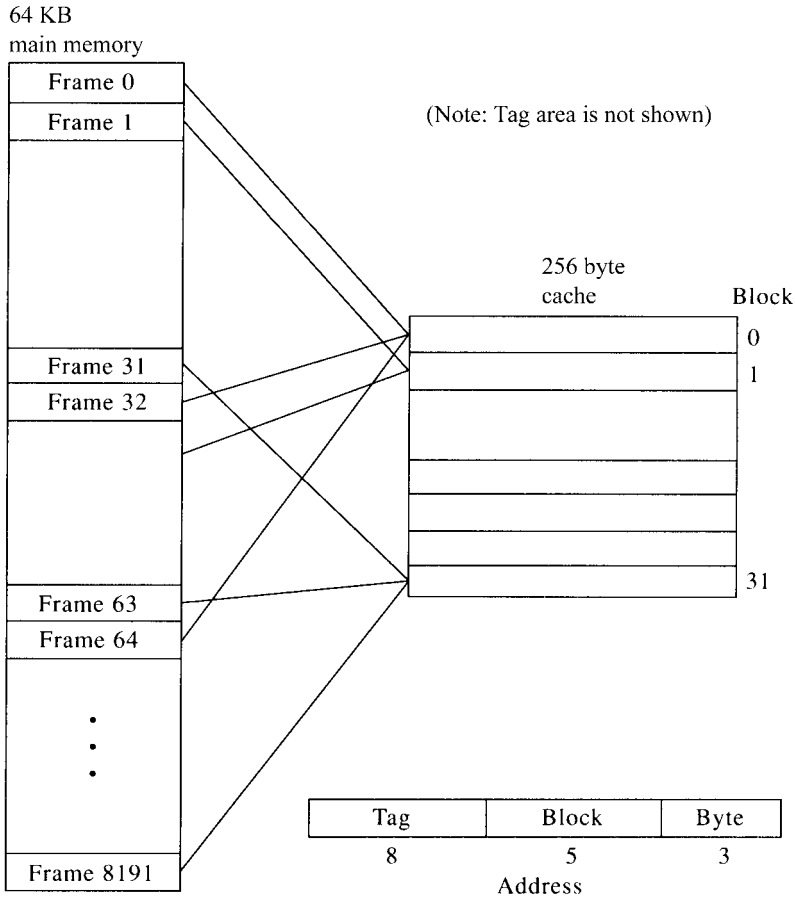



**Figure 8.8** Direct mapping (Example 8.2)

partitions, each containing $B/K$ blocks. The address partitioning for this *K-way set-associative* mapping is shown below:

| Address $A$ ($p$ bits) | Tag | Set number | Word number |
|---|---|---|---|
| | $(p-n-b+k)$ | $(b-k)$ | $n$ |

Note that this scheme is similar to the direct mapping in that it divides the main memory addresses into $2^{(b-k)}$ partitions. But, each frame can now reside in one of the $K$ corresponding cache blocks, known as a *set*. The

tag search is now limited to $K$ tags in the set. Figure 8.9 shows the details of a set-associative cache scheme. Also note the following relations:

$k = 0$ implies direct mapping.
$k = b$ implies fully associative mapping.

---

**Example 8.3** Figure 8.10 shows the 4-way set-associative scheme for the memory system of Example 8.1. Note that now the data area of the cache contains a total of 128 blocks, or 1 KB.

---

### Replacement Algorithm

When there is a cache miss, the frame corresponding to the referenced address is brought into the cache. The placement of this new block in the cache depends on the mapping scheme used by the cache mechanism. If the direct mapping is used, there is only one possible cache block the frame can occupy. If that cache block is already occupied by another frame, it is replaced by the new frame. In the case of fully associative mapping, the new frame may occupy any vacant block in the cache. If there are no vacant blocks, it will be necessary to replace one of the blocks in the cache with the new frame. In case of a $K$-way set associative mapping, if all the $K$ elements of the set into which the new frame maps are occupied, one of the elements needs to be replaced by the new frame.

The most popular replacement algorithm used replaces the *least recently used (LRU)* block in the cache. From the program locality principle, it follows that the immediate references will be to those addresses that have been referenced recently. In this case, an LRU replacement policy works well. In order to identify the LRU block, a counter can be associated with each cache block. When a block is referenced, its counter is set to zero while the counters of all the other blocks are incremented by 1. At any time, the LRU block is the one whose counter has the highest value. These counters are usually called *aging counters* since they indicate the age of the cache blocks.

The *first-in/first-out (FIFO)* replacement policy has also been used. Here, the block that is in the cache longest is replaced. To determine the block to be replaced, each time a frame is brought into the cache, its identification number is loaded into a queue. The output of the queue thus always contains the identification of the frame that entered the cache first. Although

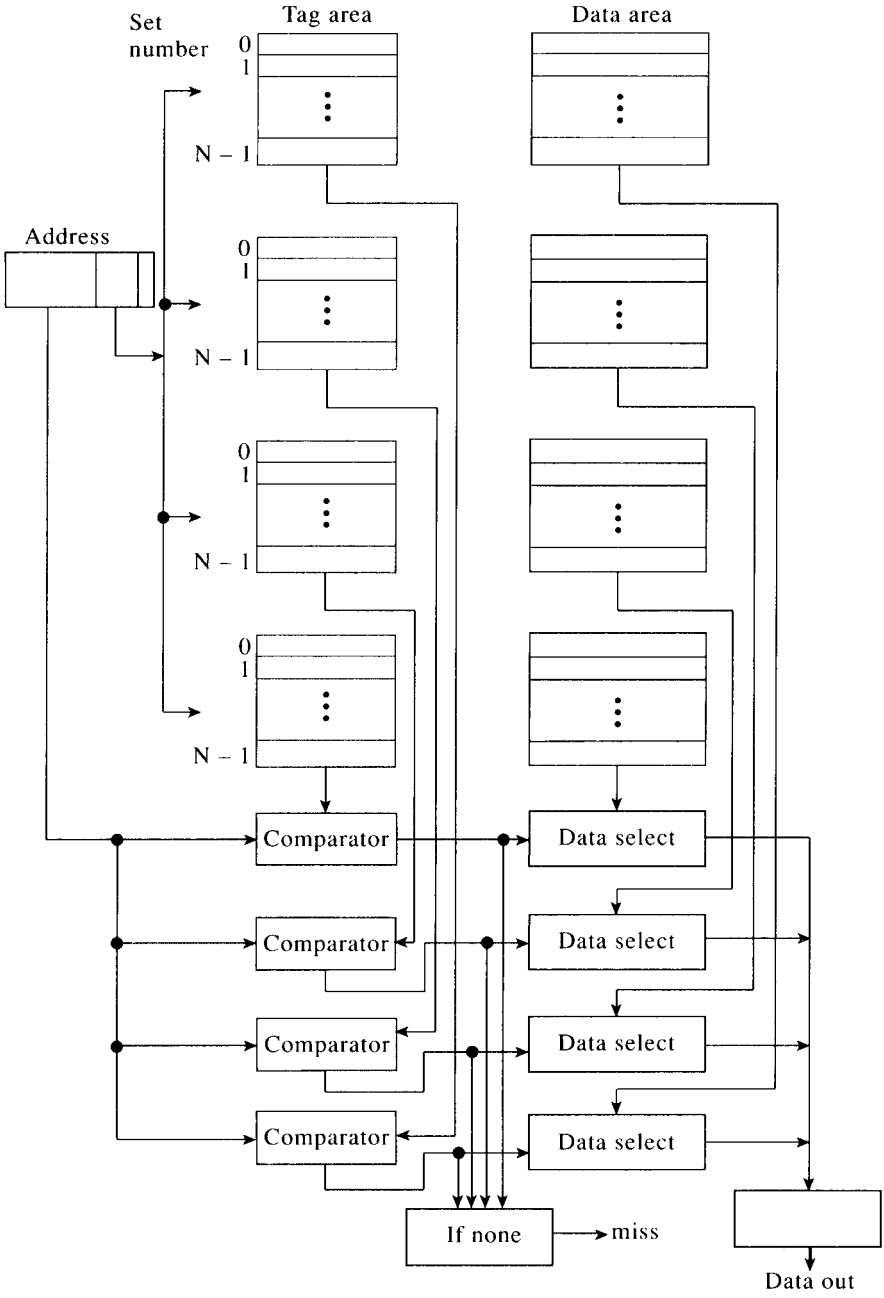



**Figure 8.9** The structure of a four-way set-associative cache with $N$ sets

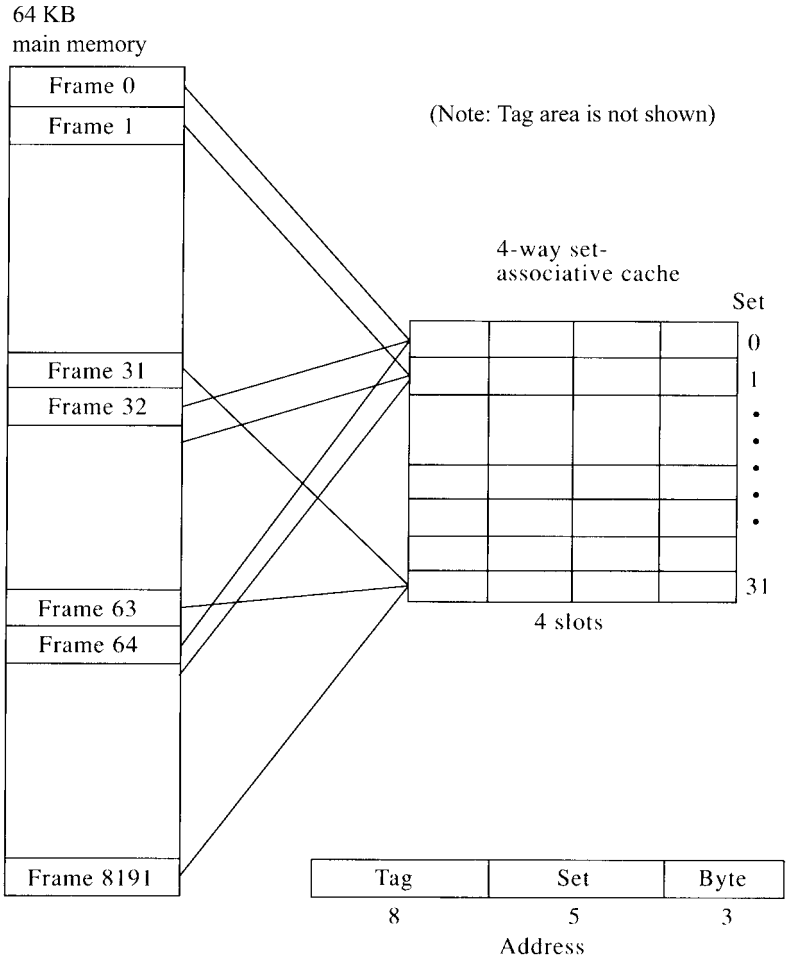



**Figure 8.10** Set-associative mapping (Example 8.3)

this mechanism is easy to implement, it has the disadvantage that under certain conditions, blocks are replaced too frequently.

Other possible policies are 1) the *least frequently used (LFU)* policy, in which the block that has experienced the least number of references in a given time period is replaced and 2) the *random* policy, in which a block among the candidate blocks is randomly picked for replacement. Simulation has shown that the performance of the random policy (which is not based on the usage characteristics of the cache blocks) is only slightly inferior to that of the policies based on usage characteristics.

### Write Operations

The description above assumed only read operations from the cache and primary memory. The processor can also access data from the cache and update it. Consequently, the data in the primary memory must also be correspondingly updated. Two mechanisms are used to maintain this data *consistency: write back* and *write through*. In the *write back* mechanism, when the processor writes something into a cache block, that cache block is tagged as a *dirty block*, using a 1-bit tag. Before a dirty block is replaced by a new frame, the dirty block is copied into the primary memory. In the *write-through* scheme, when the processor writes into a cache block, the corresponding frame in the primary memory is also written with the new data.

Obviously, the write-through mechanism has higher overhead on memory operations than the write-back mechanism. But it easily guarantees the data consistency, an important feature in computer systems in which more than one processor accesses the primary memory. Consider, for instance, a system with an I/O processor. Since the I/O processor accesses the memory in a DMA mode (not using the cache), and the central processor accesses the memory through the cache, it is necessary that the data values be consistent.

In multiple processor systems with a common memory bus, a *write-once* mechanism is used. Here, the first time a processor writes to its cache block, the corresponding frame in the primary memory is also written, thus updating the primary memory. All the other cache controllers invalidate the corresponding block in their cache. Subsequent updates to the cache affect only (local) cache blocks. The dirty blocks are copied to the main memory when they are replaced.

The write-through policy is the more common since typically the write requests are of the order of 15 to 20% of the total memory requests and hence the overhead due to write-through is not significant.

The tag portion of the cache usually contains a *valid bit* corresponding to each tag. These bits are reset to zero when the system power is turned on, indicating the invalidity of the data in all the cache blocks. As and when a frame is brought into the cache, the corresponding valid bit is set to 1.

### Performance

The average access time, $T_a$, of the memory system with cache is given by:

$$T_a = hT_c + (1 - h)T_m \tag{8.2}$$

where $T_c$ and $T_m$ are average access times of cache and primary memory respectively, $h$ is the *hit ratio*, and $(1 - h)$ is the *miss ratio*.

We would like $T_a$ to be as close to $T_c$ as possible. Since $T_m >>> T_c$, $h$ should be as close to 1 as possible. From (8.2), it follows that:

$$\frac{T_c}{T_a} = \frac{1}{h + (1 - h)T_m/T_c} \tag{8.3}$$

Typically, $T_m$ is about 5 to 10 times $T_c$. Thus, we would need a hit ratio of 0.75 or better to achieve reasonable speedup. A hit ratio of 0.9 is not uncommon in contemporary computer systems.

Progress in hardware technology has enabled very cost-effective configuration of large-capacity cache memories. In addition, as seen with the Intel Pentium architecture described in the previous chapter, the processor chip itself contains the first-level cache, and the second-level cache is configured externally to the processor chip. Some processor architectures contain separate data and instruction caches. Section 8.4 provides brief descriptions of representative cache structures.

## 8.2 SIZE ENHANCEMENT

The main memory of a machine is not usually large enough to serve as the sole storage medium for all programs and data. Secondary storage devices such as disks are used to increase capacity. However, the program and data must be in the main memory during the program execution. Mechanisms to transfer programs and data to the main memory from the secondary storage as needed during the execution of the program are thus required. Typically, the capacity of the secondary storage is much higher than that of the main memory. But the user assumes that the complete secondary storage space (i.e., *virtual address space*) is available for programs and data, although the processor can access only the main memory space (i.e., *real or physical address space*). Hence the name *virtual storage*.

In early days, when programmers developed large programs that did not fit into the main memory space, they would divide the programs into independent partitions known as *overlays*. These overlays were then brought into the main memory as and when needed for execution. Although this process appears simple to implement, it increases the program development and execution complexity and is visible to the user. Virtual memory mechanisms handle overlays in an automatic manner transparent to the user.

Virtual memory mechanisms can be configured in one of the following ways:

1. Paging systems
2. Segmentation systems, and
3. Paged-segmentation systems.

In paging systems, both real and virtual address spaces are divided into small, equal-sized partitions called *pages*. Segmentation systems use memory *segments*, which are unequal-sized blocks. A segment is typically equivalent to an overlay described above. In paged-segmentation systems, segments are divided into pages, each segment usually containing a different number of pages. We will now describe paging systems that are simpler to implement than the other two systems. Refer to the books on operating systems listed in the references section of this chapter for further details.

Consider the memory system shown in Fig. 8.11. The main memory is $2^p$ pages long. That is, there are $2^p$ *real pages* and we will denote each page slot in the main memory as a *frame*. The secondary storage consists of $2^q$ pages (i.e., there are $2^q$ *virtual pages*). Both real and virtual pages are $2^d$ words long. Also, $q >> p$. Since the user assumes that he has the complete virtual space to program in, the *virtual address* $A_v$ (i.e., the address produced by the program in referencing the memory system) consists of $(q + d)$ bits. But, the main memory or *physical address* $A_p$ consists of $(p + d)$ bits.

When a page is brought from the secondary storage into a main memory frame, the *page table* is updated to reflect the location of the page in the main memory. Thus, when the processor refers to a memory address, the $(q + d)$-bit address is transformed into the $(p + d)$-bit primary address, using the page-table information. If the referenced virtual page is not in the main memory, Table 2 (disk directory) is searched to locate the page on the disk. That is, Table 2 transforms $A_v$ into the physical address $A_{vp}$ on the disk, which will be in the form of drive number, head number, track number, and displacement within the track. The page at $A_{vp}$ is then transferred to an available real page location from which the processor accesses the data.

The operation of virtual memory is similar to that of a cache. As such, the following mechanisms are required:

1. *Address translation*: determines whether the referenced page is in the main memory or not and keeps track of the page movements in both memories.
2. *Mapping function*: determines where the pages are to be located in the main memory. The direct, fully associative and set-associative mapping schemes are used.

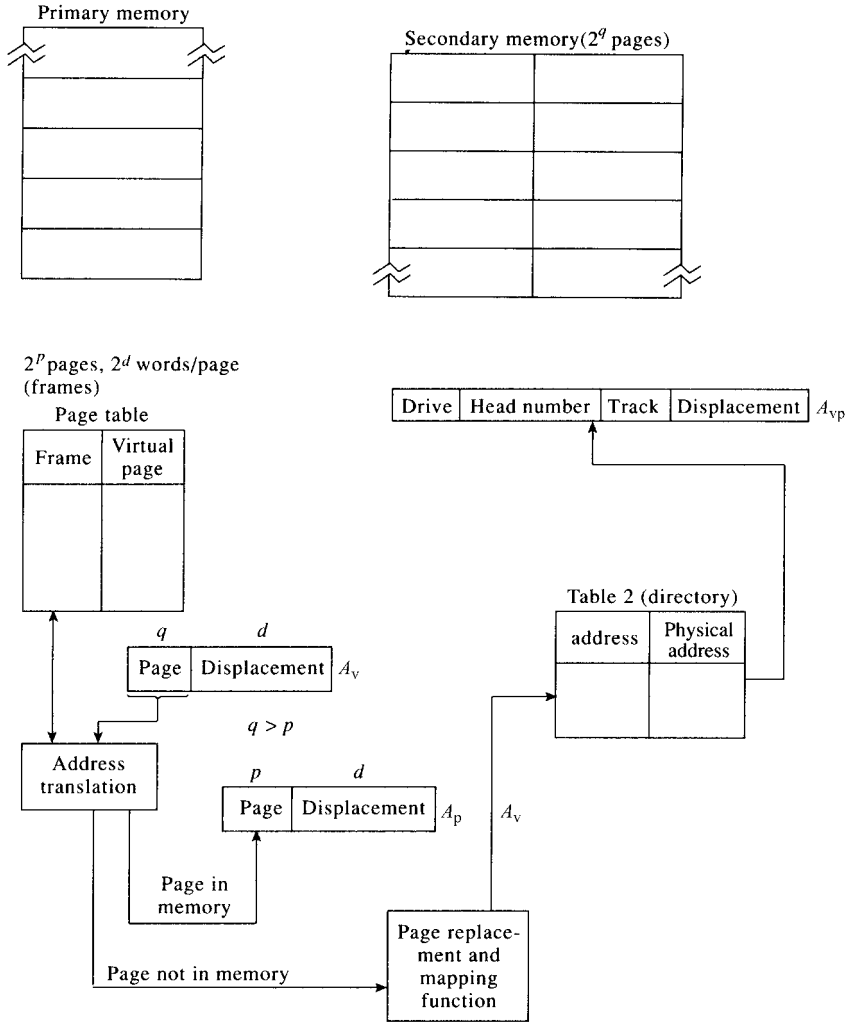



**Figure 8.11** Virtual memory system

3. *Page replacement policy*: decides which of the real pages are to be replaced. The LRU and FIFO policies are commonly used.

Although functionally similar, the cache and virtual memory systems differ in several characteristics. These differences are described below:

### Page Size

The page size in virtual memory schemes is of the order of 512 to 4K bytes, compared to 8- to 16-byte block sizes in cache mechanisms. The page size is determined by monitoring the address reference patterns of application programs, as exhibited by the program locality principle. In addition, the access parameters (such as block size) of the secondary storage devices influence the page size selection.

### Speed

When a cache miss occurs, the processor waits for the corresponding block from the main memory to arrive in the cache. In order to minimize this wait time, the cache mechanisms are implemented totally in hardware and operate at the maximum speed possible. A miss in virtual memory mechanism is termed a *page fault*. A page fault is usually treated as an operating system call in the virtual memory mechanisms. Thus, the processor is switched to perform some other task, while the memory mechanism brings the corresponding page into the main memory. The processor is then switched back to that task. Since the speed is not the main criterion, virtual memory mechanisms are not completely implemented using hardware. With the popularity of microprocessors, special hardware units called *memory management units* (MMU) that handle all the virtual memory functions are now available. One such device is described later in this chapter.

### Address Translation

The number of entries in the tag area of the cache mechanism is very small compared to the entries in the page table of a virtual memory mechanism because of the sizes of the main memory and secondary storage. As a result, maintenance of the page table and minimizing the page table search time are important considerations. We will describe the popular address translation schemes next.

Figure 8.12 shows a page-table structure for the virtual memory scheme of Fig. 8.11. The page table contains $2^q$ slots. The page field of the virtual address forms the index to the page table. When a secondary page is moved into the main memory, the main memory frame number is entered into the corresponding slot in the page table. The *residence bit* is set to 1 to indicate that the main memory frame contains a valid page. The residence bit being 0 indicates that the corresponding main memory frame is empty. The page-table search requires only one access to the main memory,

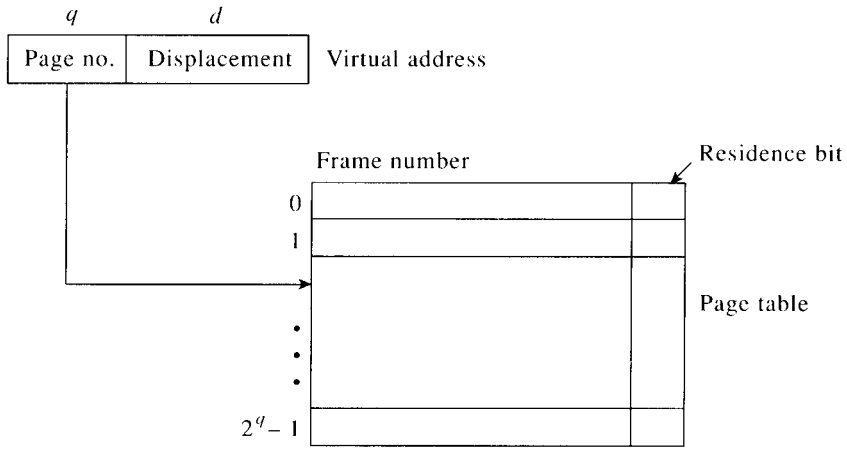



**Figure 8.12** Page-table structure with $2^q$ entries

in which the page table is usually maintained. But in this scheme, since there are only $2^p$ frames in the main memory, $2^q - 2^p$ slots in the page table are always empty. Since $q$ is large, the page table wastes the main memory space.

---

**Example 8.4** Consider a memory system with 64K of secondary memory and 8K of main memory with 1K pages. Then,

$$d = 10 \text{ bits}$$
$$(q + d) = 16 \text{ bits}$$
$$(p + d) = 13 \text{ bits}$$

(i.e., $q = 6$ and $p = 3$). There are 8 frames in the main memory. The secondary storage contains 64 pages. The page table consists of 64 entries. Each entry is 4 bits long (one 3-bit frame number and 1 residence bit).

---

Note that in order to retrieve data from the main memory, two accesses are required (the first to access the page table and the second to retrieve the data) when there is no page fault. If there is a page fault, then the time required to move the page into the main memory needs to be added to the access time.

Figure 8.13 shows a page-table structure containing $2^p$ slots. Each slot contains the virtual page number and the corresponding main memory frame number in which the virtual page is located. In this structure, the page table is first searched to locate the page number in the virtual address

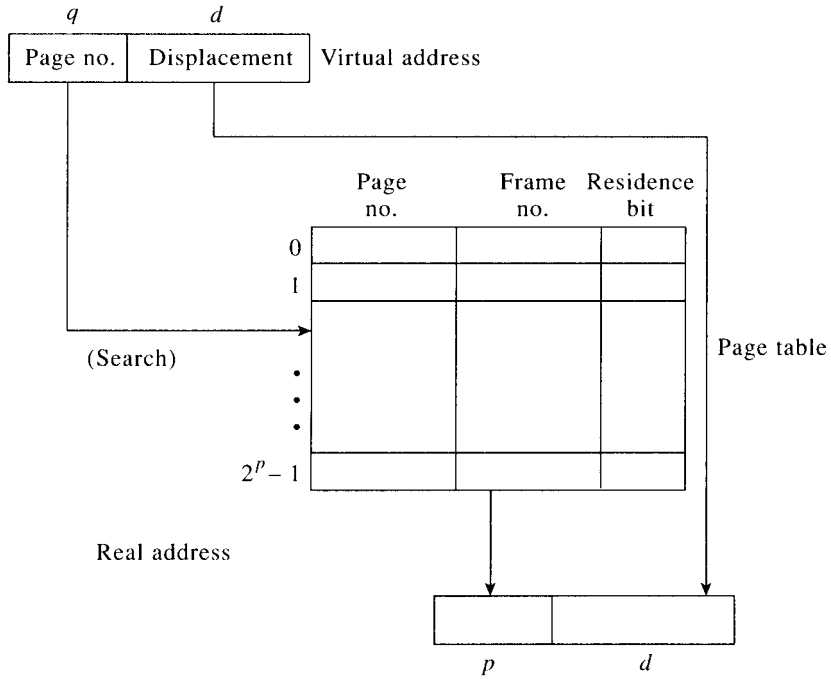



**Figure 8.13** Page-table structure with $2^p$ entries

reference. If a matching number is found, the corresponding frame number replaces the $q$ bit virtual page number. This process requires, on average, $2^p/2$ searches through the page table.

---

**Example 8.5** For the memory system of Example 8.4, a page table with only 8 entries is needed in this scheme. Each entry consists of 10 bits (a residence bit, a 3-bit main memory frame number and a 6-bit secondary storage page number).

---

Figure 8.14 shows a scheme to enhance the speed of page-table search. Here, the most recently referenced entries in the page table are maintained in a fast memory (usually an associative memory) called the *translation lookaside buffer* (TLB). The search is initiated both in TLB and the page table simultaneously. If a match is found in the TLB, the search in the page table is terminated. If not, the page table search continues.

Consider a 64 KB main memory with 1 KB page size. That is, $d = 10$ and $p = 6$, since the main memory address is 16 bits long. The page table

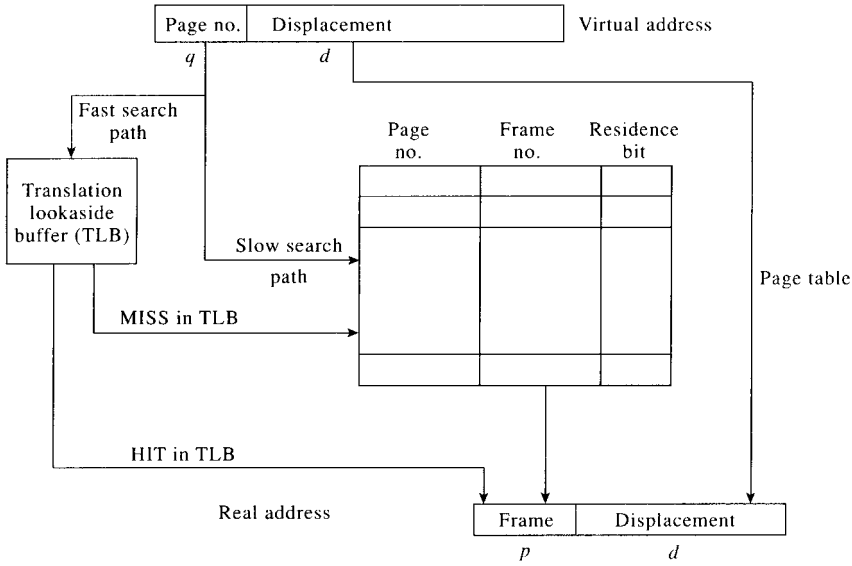



**Figure 8.14** Translation lookaside buffer

contains only $2^6 = 64$ entries and can easily be maintained in the main memory. If the main memory capacity is increased to 1 MB, then the page table would contain 1K entries, which is a significant portion of the main memory. Thus, as the main memory size grows, the size of the page table poses a problem. In such cases, the page table could be maintained in the secondary storage. The page table is then divided into several pages. It is brought into the main memory one page at a time to perform the comparison.

Figure 8.15 shows the *paged segmentation* scheme of virtual memory management. In this scheme, the virtual address is divided into a segment number, a page number, and a displacement. The segment number is an index into a segment table whose entries point to the base address of the corresponding page tables. Thus there is a page table for each segment. The page table is searched in the usual manner to arrive at the main memory frame number.

## 8.3 ADDRESS EXTENSION

In our description of the virtual memory schemes, we have assumed that the capacity of the secondary storage is much higher than that of the main memory. Also, we assumed that all memory references by the CPU are in

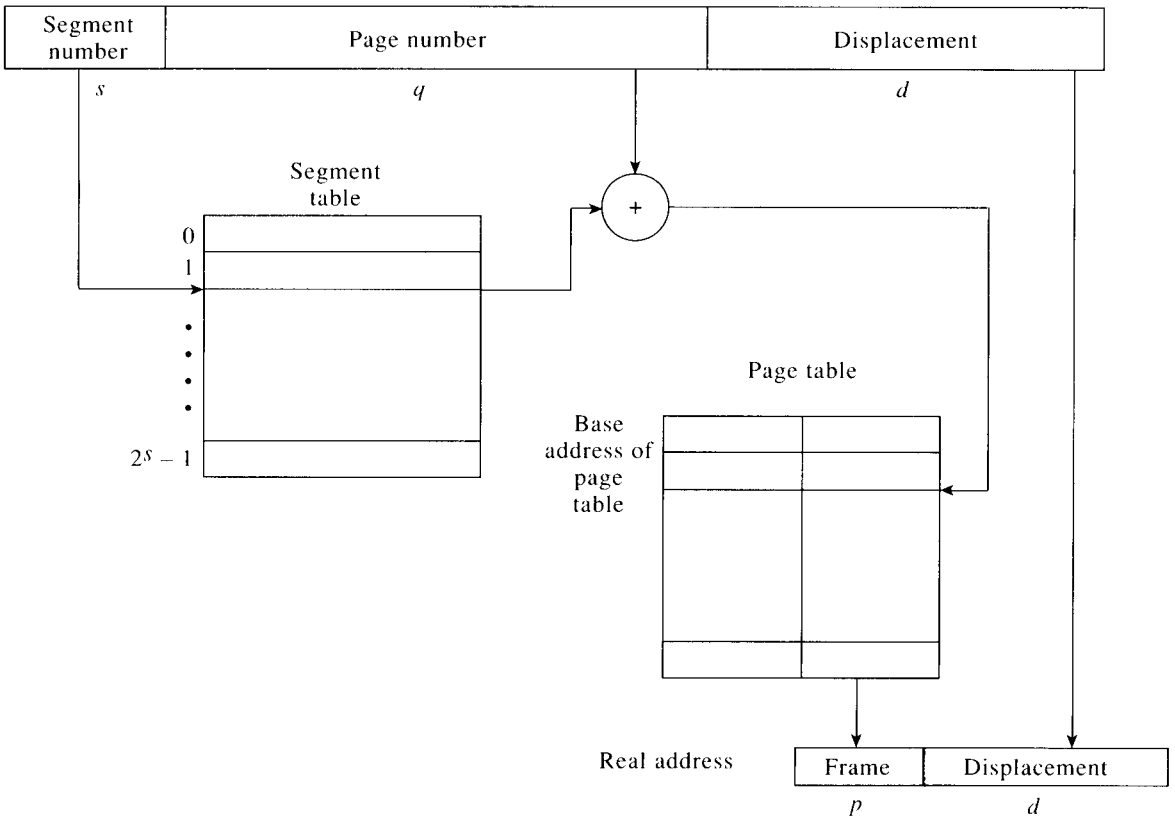



**Figure 8.15** Paged segmentation scheme

terms of virtual addresses. In practice, the addressing range of the CPU is limited, and the direct representation of the virtual address as part of the instruction is not possible.

Consider the instruction format of ASC for instance. Here, the direct addressing range is only the first 256 words of the memory. This range was extended to 64 Kwords by indexing and indirect addressing. Suppose the secondary storage is 256 Kwords long, thus requiring 18 bits in the virutal address. Since ASC can only represent a 16-bit address, the remaining 2 address bits must be generated by some other mechanism. One possibility is to output the 2 most significant address bits into an external register using the WWD command and the lower 16 bits of the address on the address bus, thus enabling the secondary storage mechanism to reconstruct the required 18-bit address, as shown in Fig. 8.16. In order to make this scheme work, the assembler should be modified to assemble all programs in an 18-bit address space where the lower 16 bits are treated as displacement addresses from a 2-bit page (or bank) address maintained as a constant with respect to each program segment. During execution, the CPU executes a WWD to output the page address into the external register as the first instruction in the program segment.

The address extension concept illustrated in Fig. 8.16 usually termed *bank switching*. In fact, this scheme can be generalized to extend the address further. As shown in Fig. 8.17, the most significant $m$-bits of the address can be used to select one of the $2^m$ $n$-bit registers. If the address bus is $p$ bits wide, then the virtual address will be $(n + p)$ bits long. In this description, we have assumed that the data bus is used to output the $m$-bit address portion. In practice, it need not be so, and the address bus can be time-multiplexed to transfer the two address portions. The DEC PDP-11/23 system uses this address extension scheme. As can be seen in Fig. 8.18, processors that use either paged or base-displacement addressing modes accommodate the address extension naturally.

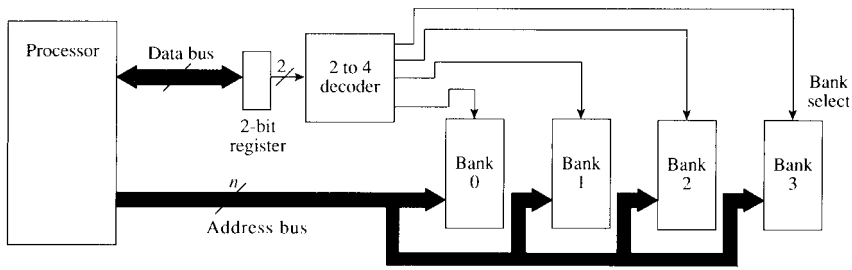



**Figure 8.16** Bank switching

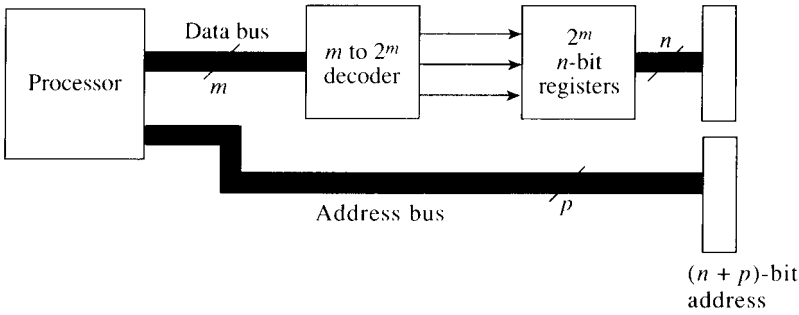



**Figure 8.17** Address extension

## 8.4 EXAMPLE SYSTEMS

A variety of structures have been used in the implementation of memory hierarchies in modern computer systems. In early systems, it was common to see the cache mechanism implemented completely using hardware, while the virtual memory mechanism was software-intensive. With advances in hardware technology, memory management units (MMU) became common. Some MMUs managed only the cache level of the hierarchy, while the others managed both the cache and virtual levels. It is now common to see most of the memory management hardware implemented on the processor chip itself. In this section we will provide a brief description of three memory systems. We first expand on the description of the Intel Pentium processor structure of Chapter 7, to show its memory management details. This is followed by the hardware structure details of the Motorola 68020 processor. The final example describes the memory management aspects of the Sun Microsystems' System-3.

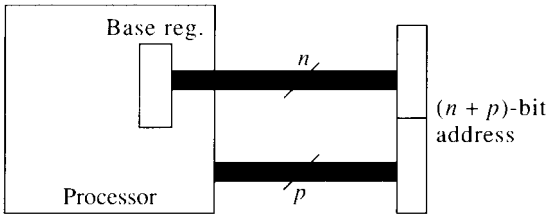



**Figure 8.18** Address extension with base register

### 8.4.1 Memory Management in Intel Processors

This section is extracted from the Intel Architecture Software Developer's Manuals listed in the References section of this chapter. An operating system designed to work with a processor uses the processor's memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Programs usually do not address physical memory directly when they employ the processor's memory management facilities. The processor's addressable memory space is called *linear address space*. Segmentation provides a mechanism for dividing the linear address space into smaller protected address spaces called *segments*. Segments in Intel processors are used to hold the code, data, and stack for a program or to hold system data structures. Each program has its own set of segments if more than one program (or task) is running on a processor. The processor enforces the boundaries between these segments and insures that one program does not interfere with the execution of another program by writing into the other program's segments. The segmentation also allows typing of segments so that the operations that are performed on a particular type of segment can be restricted. All of the segments within a system are contained in processor's linear address space. The Intel architecture supports either direct physical addressing of memory or virtual memory through paging. A linear address is treated as a physical address when physical addressing is used. All the code, data, stack and system segments are paged with only the most recently accessed pages being held in physical memory when paging is used.

Any program running on an Intel processor is given a set of resources for executing instructions and for storing code, data, and state information. They include an address space of up to $2^{32}$ bytes, a set of general data registers, a set of segment registers (see Chapter 7 for details), and a set of status and control registers. Only the resources that are associated with the memory management will be described in this section.

Intel architecture uses three memory modes: *flat*, *segmented*, and *real-address mode*. The linear address space appears as a single and continuous address space to a program with the flat memory model as shown in Fig. 8.19. The code, data and the procedure stacks are all contained in this address space. The linear address space for this model is byte addressable. The size of the linear address space is 4 GB.

With the segmented memory model, the linear address space appears to be a group of independent segments, as shown Fig. 8.20. The code, data and stacks are stored in separate segments when this model is used. The program also needs to issue a logical address to address a byte in a segment

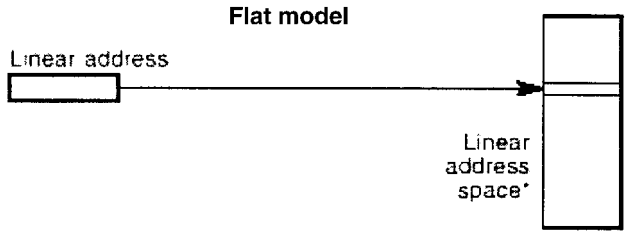



**Figure 8.19** Flat model

when this model is used. The segment selector is used here to identify the segment to be accessed and the offset is used to identify a byte in the address space of the segment. A processor can address up to 16,383 segments of different sizes and types. Each segment can be as large as $2^{32}$ bytes.

The real-address mode model (Fig. 8.21) uses a specific implementation of segmented memory in which the linear address space for the program and the operating system consists of an array of segments of up to 64 KB each. The maximum size of the linear address space in real-address mode is $2^{20}$ bytes.

A processor can operate in three operating modes: *protected*, *real-address* and *system management*. The processor can use any of the memory models when in the protected mode. The use of memory models depends on the design of the operating system. Different memory models are used when multitasking is implemented. The processor only supports the real-address mode memory model when the real-address mode is used. The processor switches to a separate address space called the system management RAM (SMRAM) when the system management mode is used. The memory model used to address bytes in this address space is similar to the real-address mode model.

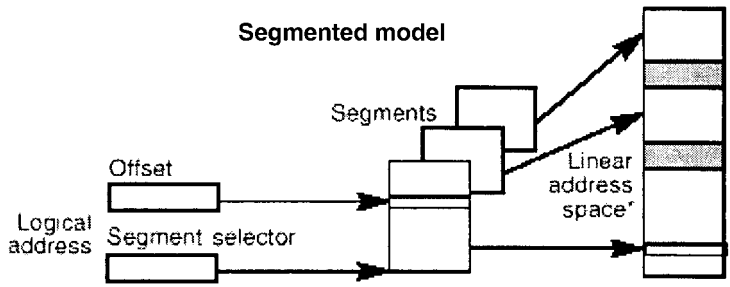



**Figure 8.20** Segmented model

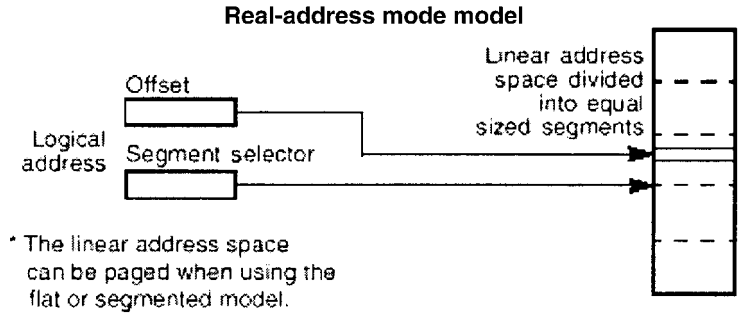

Real-address mode model
Linear address space divided into equal sized segments
Offset
Logical address
Segment selector
* The linear address space can be paged when using the flat or segmented model.

**Figure 8.21** Real-address mode model

The Intel processor provides four memory-management registers: the global descriptor table register (GDTR), the local descriptor table register (LDTR), the interrupt descriptor table register (IDTR), and the task register (TR). These registers specify the locations of the data structures that control segmented memory management. Figure 8.22 shows the details.

The GDTR is 48 bits long and is divided into two parts: a 16-bit table limit and a 32-bit linear base address. The 16-bit table limit specifies the number of bytes in the table and the 32-bit base address specifies the linear address of byte 0 of GDT. There are two instructions that are used to load (LGDT) and store (SGDT) the GDTR. The IDTR is also 48 bit long and is divided into two parts: 16-bit table limit and 32-bit linear base address. The 16-bit table limit specifies the number of bytes in the table and the 32-bit base address specifies the linear address of byte 0 of IDT. LIDT and SIDT are used to load and store this register. The LDTR has system segment registers and segment descriptor registers. The system segment registers are 16-bit long, and they are used to select segments. The segment limit of

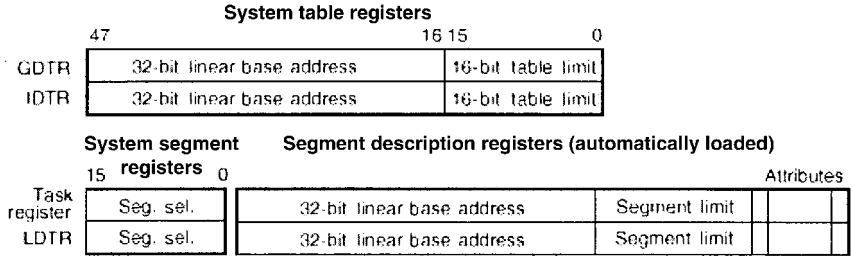



**Figure 8.22** Memory management registers

TR is automatically loaded along with the task register when a task switch occurs.

There are five control registers: CR0, CR1, CR2, CR3, and CR4, which determine the operating mode of the processor and the characteristics of the currently executing tasks.

The Intel Pentium architecture supports caches, translation lookaside buffers (TLBs), and write buffers for temporary on-chip storage of instructions and data (See Fig. 8.23). There are two levels of cache: level 1 (L1) and level 2 (L2). The L1 cache is closely coupled to the instruction fetch unit of the processor, and the L2 cache is closely coupled to the L1 cache through the processor's system bus. For the Intel 486 processor, there is only one cache for both instruction and data. The Pentium and its subsequent processor series have two separate caches, one for instruction and the other for data. The Intel Pentium processor's L2 cache is external to the processor package, and it is optional. The L2 cache is a unified cache for storage of both instructions and data.

A linear address contains two items: a segment selector and an offset. A segment selector is a 16-bit identifier for a segment. It usually points to the segment descriptor that defines the segment. A segment selector contains three items: index, T1 (table indicator) flag, and requested privilege level (RPL). The index is used to select one of 8192 ($2^{32}$) descriptors in the GDT or LDT. The T1 flag is used to select which descriptor table to use. The RPL is used to specify the privilege level of the selector. Figure 8.24 shows how these items are arranged in a segment selector.

For the processor to access a segment, the segment must be loaded into the appropriate segment register. The processor provides six registers for holding up to six segment selectors. Each of these segment registers supports a specific kind of memory reference (code, stack, or data). The code-segment (CS), data-segment (DS), and stack-segment (SS) registers

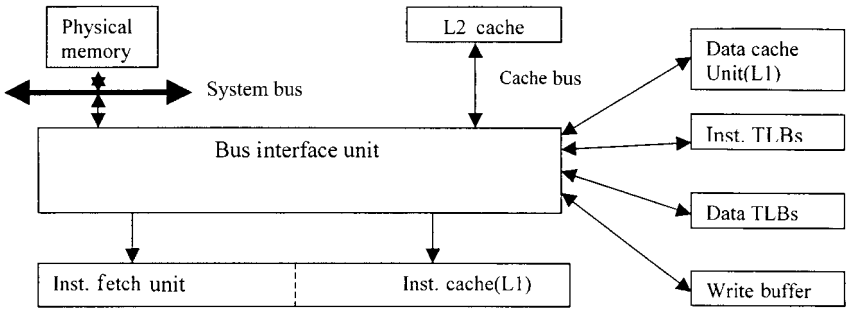



**Figure 8.23** Intel cache architecture

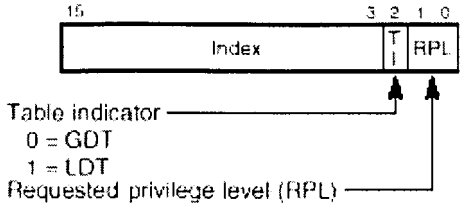



**Figure 8.24** Segment selector

must be loaded with valid segment selectors for any kind of program execution to take place. However, the processor also provides three additional data-segment registers: ES, FS, and GS. These registers can be used to make additional data segments available to the programs.

A segment register has two parts: a visible part and a hidden part. The processor loads the base address, segment limit, and access information into the hidden part and, segment selector into the visible part, as shown in Fig. 8.25.

A segment descriptor is an entry in a GDT or LDT. It provides the processor with the size and location of a segment as well as access control and status information. In the Intel system, compilers, linkers, loaders, or the operating system typically create the segment descriptors. Figure 8.26 shows the general descriptor format for all types of Intel segment descriptors.

A segment descriptor table is a data structure in the linear address space and contains an array of segment descriptors. It can contain up to 8192 ($2^{32}$) 8-byte descriptors. There are two kinds of descriptor tables: the global descriptor table (GDT) and the local descriptor tables (LDT). Each Intel system must have one GDT defined and one or more LDT defined. The base linear address and limit of the GDT must be loaded into the GDTR. The format is shown in Fig. 8.27.

| Visible part | Hidden part | |
|---|---|---|
| Segment selector | Base address, limit, access information | CS |
| | | SS |
| | | DS |
| | | ES |
| | | FS |
| | | GS |

**Figure 8.25** Segment register

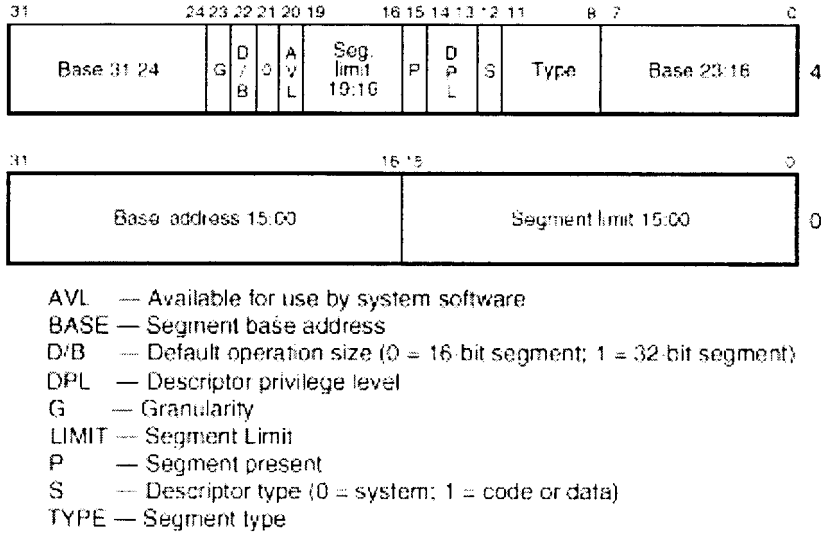



**Figure 8.26** Segment descriptor

Based on the segmentation mechanism used, a wide variety of system designs are possible. The Intel system design varies from flat models that make only minimal use of segmentation to protect programs to multi-segmented models that use segmentation to create a powerful operating environment. A few examples of how segmentation is used in the Intel systems to improve memory management performance are provided in the following paragraphs.

The basic flat model is the simplest memory model for a system. The detail of this model is shown in Fig. 8.28. It gives the operating system and programs access to a continuous unsegmented address space. The Intel system implements this model in a way that hides the segmentation mechanism of the architecture from both system designer and the application programmer in order to give the greatest extensibility.

In order to implement this model, the system must create at least two segment descriptors. One is for referencing a code segment and the other is for referencing a data segment. Both of these segments are mapped to the entire linear space. Both segment descriptors point to the same base address value of 0 and have the same segment limit of 4 GB.

The protected flat model is similar to the basic flat model. The only difference is in the segment limits. In this model, the segment limits are set to include only the range of addresses for which physical memory actually

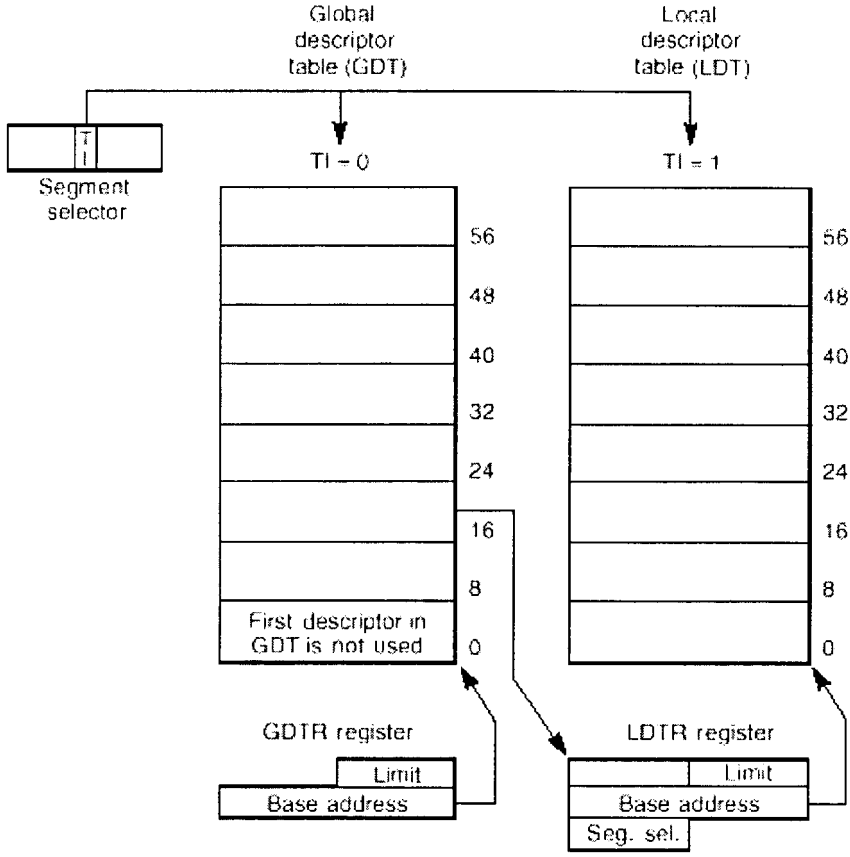



**Figure 8.27** Global and local descriptor tables

exists. This model is shown in Fig. 8.29. This model can provide better protection by adding a simple paging structure.

The multi-segmented model uses the full capabilities of the segmentation mechanism to provide the protection of the code, data structures, and tasks. The model is shown in Fig. 8.30. In this model, each program is given its own table of segment descriptors and its own segments. The segments are completely private to their assigned programs or shared among programs. As shown in the figure, each segment register has a separate segment descriptor associated with it, and each segment descriptor points to a separate segment. Access to all segments and to the execution environments of individual programs running on the Intel system is controlled by hardware.

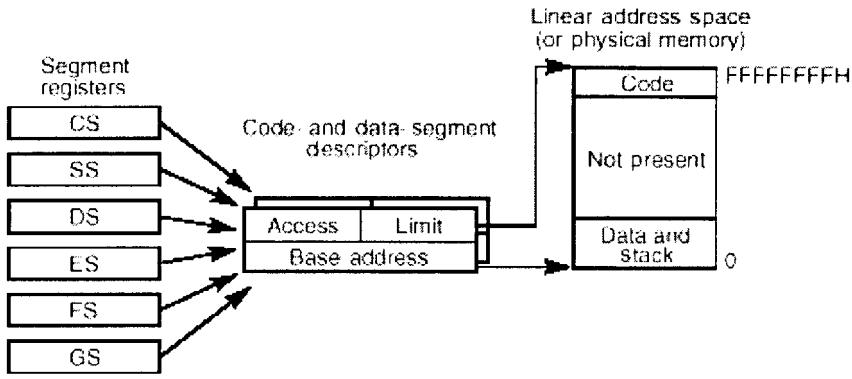



**Figure 8.28** Basic flat model

Four control registers facilitate paging options. CR0 contains system control flags that control the operating mode and states of the processor. Bit 31 of CR0 is the paging (PG) flag. The PG flag enables the page-translation mechanism. This flag must be set if the processor is implementing a demand-paged virtual memory system or if the operating system is designed to run more than one program (or task) in virtual 8086 mode. CR2 contains the linear address that causes a page fault. CR3 contains the physical address of the base of the page directory. It is referred to as the page directory base register (PDBR). Only the 20 most significant bits of the PDBR are specified – the lower 12 bits are assumed to be 0. Bit 4 of CR4 is the page size extensions (PSE) flag. The PSE flag enables a larger page size of 4 MB.

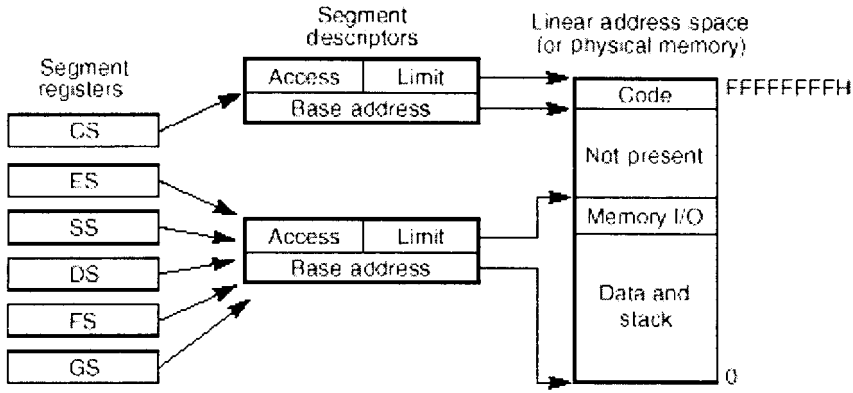



**Figure 8.29** Protected flat model

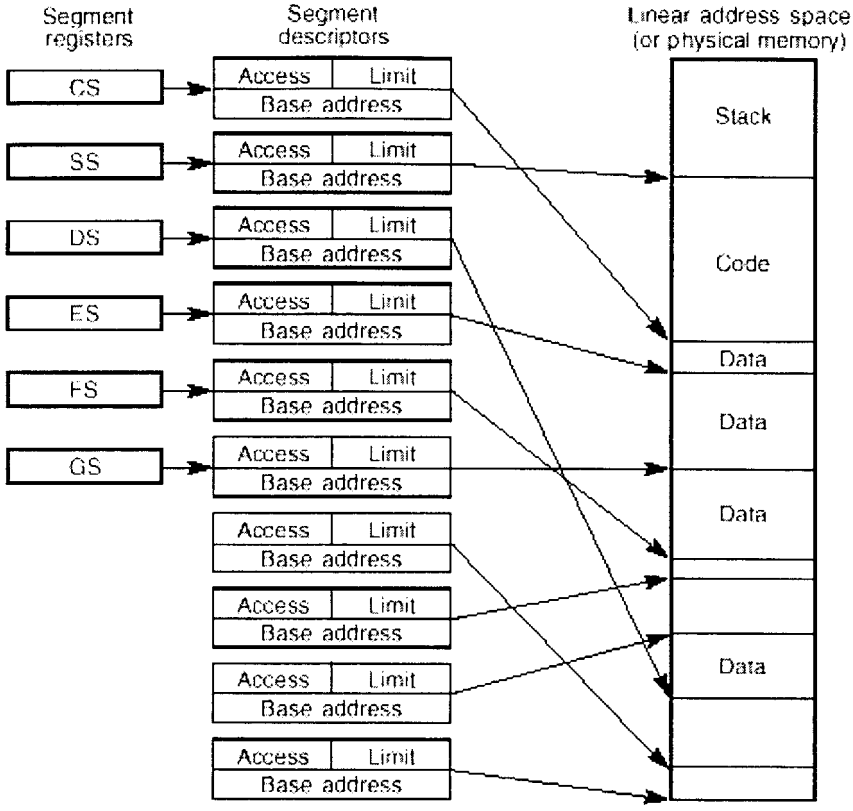



**Figure 8.30** Multi-segment model

When the PSE flag is not set, the more common page length of 4 KB is used. Figure 8.31 shows the various configurations that can be achieved by setting the PG and PSE Flags as well as the PS (page size) flag.

When paging is enabled, the processor uses the information contained in three data structures to translate linear address into physical addresses. The first data structure is a page directory. This is an array of 32-bit page-directory entries (PDEs) contained in a 4 KB page. Up to 1024 page-directory entries can be held in a page directory. The second data structure is the page table. This is an array of 32-bit page-table entries (PTEs) contained in a 4 KB page. Up to 1024 page-table entries can be held in a page table. The third data structure is the page. This is either a 4 KB or 4 MB flat address space depending on the setting of the PS flag. Figures 8.32 and 8.33 show the formats of page-directory and page-table entries when 4 KB pages and 4 MB pages are used.

| PG flag, CR0 | PSE flag, CR4 | PS flag, PDE | Page size | Physical address size |
|---|---|---|---|---|
| 0 | X | X | — | Paging Disabled |
| 1 | 0 | X | 4 KB | 32 bits |
| 1 | 1 | 0 | 4 KB | 32 bits |
| 1 | 1 | 1 | 4 MB | 32 bits |

**Figure 8.31** Page and physical address sizes

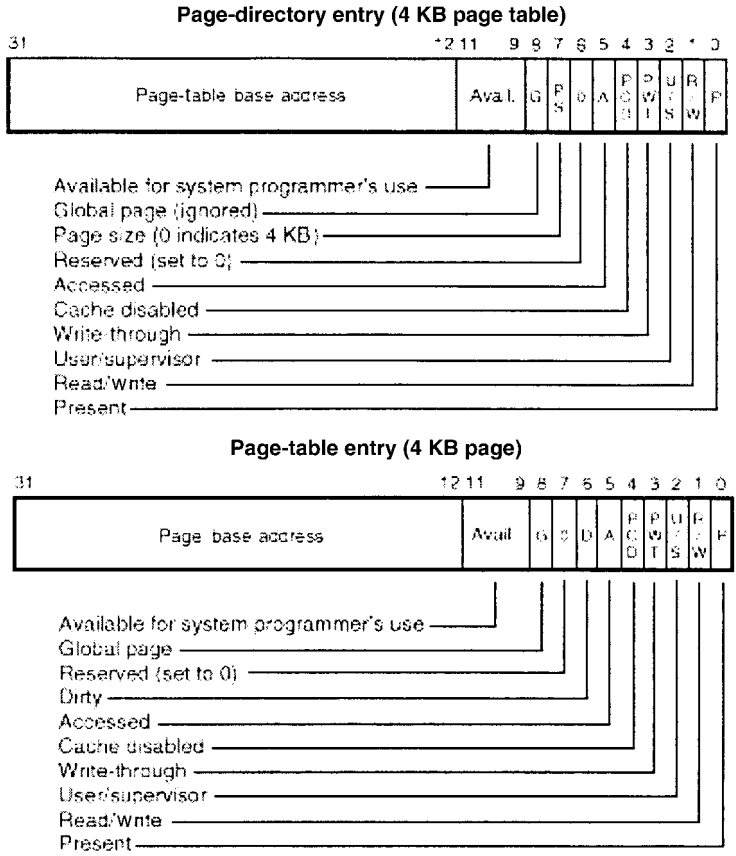



**Figure 8.32** Format of page-directory and page-table entries for 4 KB pages

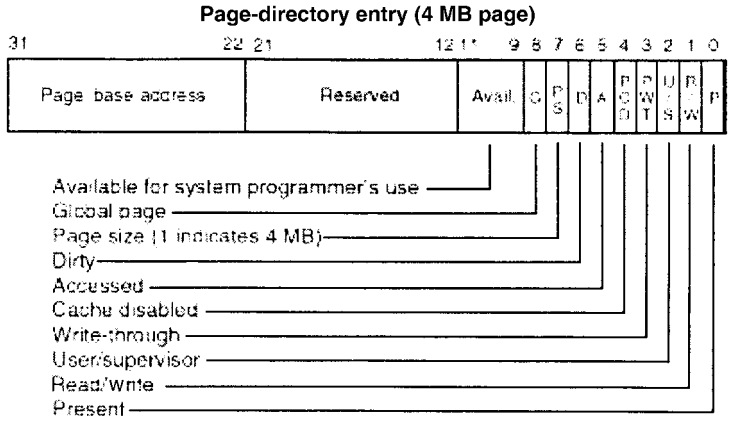



**Figure 8.33** Format of page-directory entry for 4 MB pages

The only difference in the above formats is that the 4 MB format does not utilize page tables. As a result, there are no page table entries. The functions of the flags and fields in the above entries are as follows:

- Page base address (bits 12–31/22–31): for 4 KB pages, this specifies the physical address of the first byte of a 4 KB page in the page table entry, and the physical address of the first byte of a page table in the page-directory entry. For 4 MB pages, this specifies the physical address of the first byte of a 4 MB page in the page directory. Only bits 22–31 are used. The rest of the bits are reserved and must be set to 0.
- Present (P) flag (bit 0): This indicates whether the page or the page table being pointed to by the entry is currently loaded in physical memory. When this flag is clear, the page is not in memory. If the processor attempts to access the page, it generates a page-fault exception.
- Read/write (R/W) flag (bit 1): This specifies the read/write privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When the flag is clear, the page is read only, and when the flag is set, the page can be read or written to.
- User/supervisor (U/S) flag (bit 2): This specifies the user/supervisor privileges for a page or a group of pages. When this flag is clear, the page is assigned the supervisor privilege level, and when the flag is set, the page is assigned the user privilege level.

- Page-level write-through (PWT) flag (bit 3): This controls the write-through or write-back caching policy of individual pages or page tables.
- Page-level cache disable (PCD) flag (bit 4): This controls whether or not individual pages or page tables can be cached.
- Accessed (A) flag (bit 5): This indicates whether or not a page or page table has been accessed.
- Dirty (D) flag (bit 6): This indicates whether or not a page has been written to.
- Page size (PS) flag (bit 7): This determines the page size. When this flag is clear, the page size is 4 KB and the page-directory entry points to a page table. When the flag is set, the page size is 4 MB for normal 32-bit addressing and the page-directory entry points to a page. If the page-directory entry points to a page table, all the pages associated with that page table will be 4 KB long.
- Global (G) flag (bit 8): This indicates a global page when set.
- Available-to-software bits (bit 9–11): These bits are available for use by software.

Translation lookaside buffers are on-chip caches that the processor utilizes to store the most recently used page-directory and page-table entries. The Pentium processor has separate TLBs for data and instruction caches as well as for 4 KB and 4 MB page sizes. Paging is most performed using the contents of the TLBs. If the TLBs do not contain the translation information for a requested page, then bus cycles to the page directory and page tables in memory are performed.

Figure 8.34 shows the page directory and page-table hierarchy when mapping linear addresses to 4 KB pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to $2^{20}$ pages, which spans a linear address space of $2^{32}$ bytes (4 GB). To select the various table entries, the linear address is divided into three sections. Bits 22–31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table. Bits 12–21 provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory. Bits 0–11 provide an offset to a physical address in the page.

Figure 8.35 shows the use of a page directory to map linear addresses to 4 MB pages. The entries in the page directory point to 4 MB pages in physical memory. Page tables are not used for 4 MB pages. These page sizes are mapped directly from one or more page-directory entries.

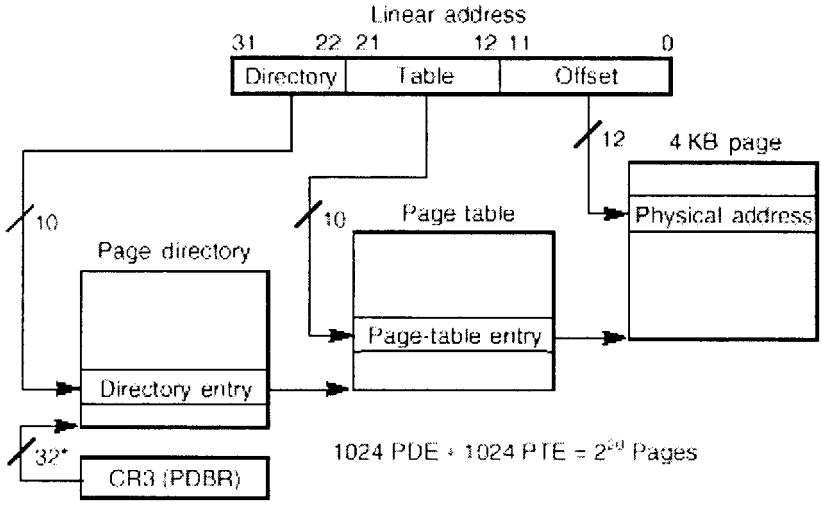



**Figure 8.34** Linear address translation (4 KB pages)

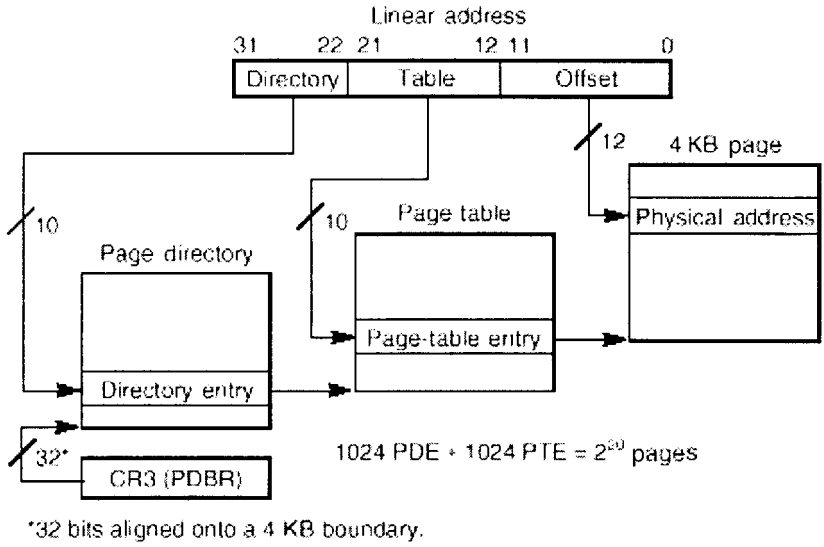



**Figure 8.35** Linear address translation (4 MB pages)

The PDBR points to the base of the page directory. Bits 22–31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a 4 MB page. Bits 0–21 provide an offset to a physical address in the page.

As explained previously, when the PSE flag in CR4 is set both 4 MB pages and page tables for 4 KB pages can be accessed from the same page directory. Mixing 4 KB and 4 MB pages can be useful. For example, the operating system or executive's kernel can be placed in a large page to reduce TLB misses and thus improve overall system performance. The processor to maintain 4 MB page entries and 4 KB page entries uses separate TLBs. Due to this, placing often used code such as the kernel in a large page frees up 4 KB page TLB entries for application programs and tasks.

The Intel architecture supports a wide variety of approaches to memory management using the paging and segmentation mechanisms. There is no forced correspondence between the boundaries of pages and segments. A page can contain the end of one segment and the beginning of another. Likewise, a segment can contain the end of one page and the beginning of another.

Segments can be mapped to pages in several ways. One such implementation is demonstrated in Fig. 8.36. This approach gives each segment its own page table, by giving each segment a single entry in the page directory which provides the access control information for paging the entire segment.

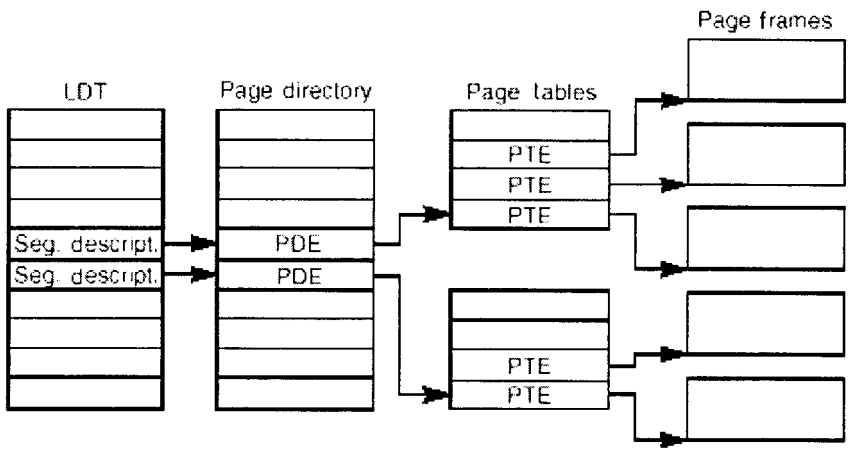



**Figure 8.36** Memory management convention that assigns a page table to each segment

### 8.4.2 Motorola 68020 Memory Management

Figure 8.37 shows a block diagram of MC68020 instruction cache. Data is not cached in this system. The cache contains 64 entries. Each entry consists of a 26-bit tag field and a 32-bit data (instruction) field. The tag field contains the most significant 24 bits (A8–A31) of the memory address, a valid bit and the function code bit FC2. FC2 = 1 indicates a supervisory mode, and FC2 = 0 indicates a user mode of operation. Address bits A2 through A7 are used to select one of the 64 entries in the cache. Thus the cache is direct-mapped. The data field can hold two 16-bit instruction words.

During the cache access, the tag field from the selected entry is first matched with A8–A31. If there is a match and the valid bit is 1, A1 selects the upper or lower 16-bit word from the data area of the selected entry. If there is no match or if the valid bit is 0, a miss occurs. Then, the 32-bit

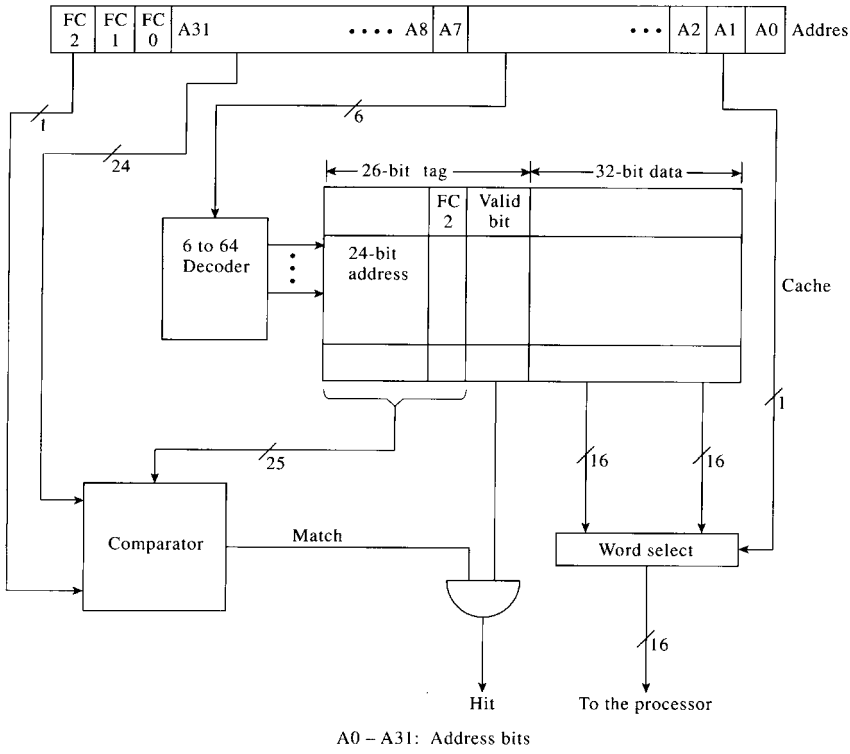



**Figure 8.37** MC68020 cache organization

instruction from the referenced address is brought into the cache and the valid bit is set to 1.

In this system, the on-chip cache access requires two cycles, while the off-chip memory access requires three cycles. Thus, during hits, the memory access time is reduced considerably. In addition, the data can be accessed simultaneously during the instruction fetch.

### 8.4.3 Sun-3 System Memory Management

Figure 8.38 shows the structure of the Sun-3 workstation, a microcomputer system based on MC68020. In addition to the processor, the system uses a floating point coprocessor (MC68881), a memory management unit, a cache unit, 4 or 8 MB of memory, an Ethernet interface for networking, a high-resolution monitor with a 1-Mbyte display memory, keyboard and mouse ports, two serial ports, bootstrap EPROM, ID PROM, configuration EEROM, time-of-day clock, and memory bus interface. External devices are interfaced to 3/100 and 3/200 systems through a VME bus interface. We will restrict this description to the memory management aspects of the system.

Although MC68020 has a memory management coprocessor (MC68851), Sun Microsystems chose to use an MMU of their own design. This MMU supports mapping of up to 32 Mbytes of physical memory at a time out of a total virtual address space of 4 gigabytes.

The virtual address space is divided into four physical (i.e., real) address spaces: one physical memory and three I/O device spaces. The physical address space is further divided into 2048 segments, with each segment containing up to 16 pages. The page size is 8 KB. The segment and page tables are stored in a high-speed memory attached to the MMU rather than in the main memory; this speeds up the address translation since page faults do not occur during the segment and page-table accesses.

The MMU contains a 3-bit *context register*. The address translation mechanism is shown in Fig. 8.39. The contents of the context register are concatenated with the 11-bit segment number. These 14 bits are used as an index to the segment table to fetch a page-table pointer. The page-table pointer is then concatenated to the 4-bit virtual page number and used to access a page descriptor from the page table. The page table yields a 19-bit physical page number. This is concatenated to the 13-bit displacement from the virtual address to derive the 32-bit physical address that allows an addressing range of 4 GB in each of the four address spaces.

In addition to the page number, the page-table entries contain protection, statistics, and page type fields. The protection field contains the following bits:

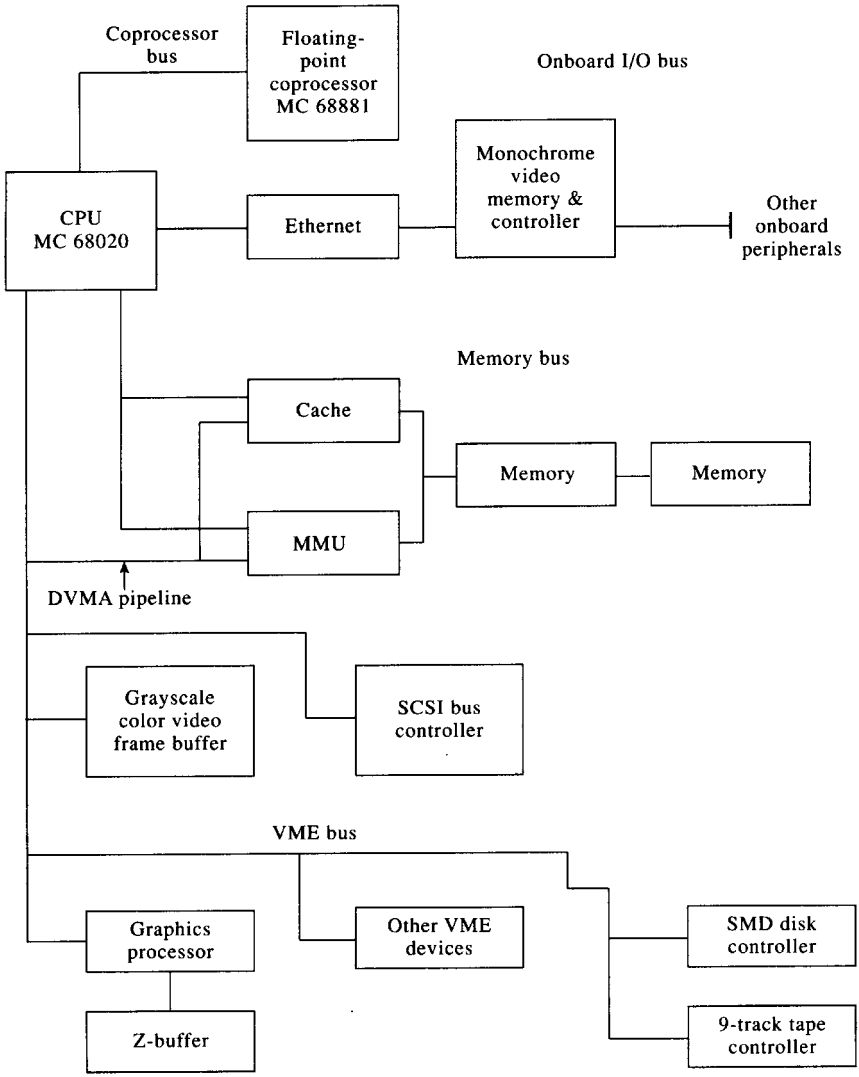



**Figure 8.38** Sun-3 system block diagram (Courtesy of Sun Microsystems Inc.)

*Valid bit* This bit indicates that the page is valid. If it is not set, any access to the corresponding page results in a page fault.

*Write enable bit* When set, this bit indicates that a write operation is allowed on the page.

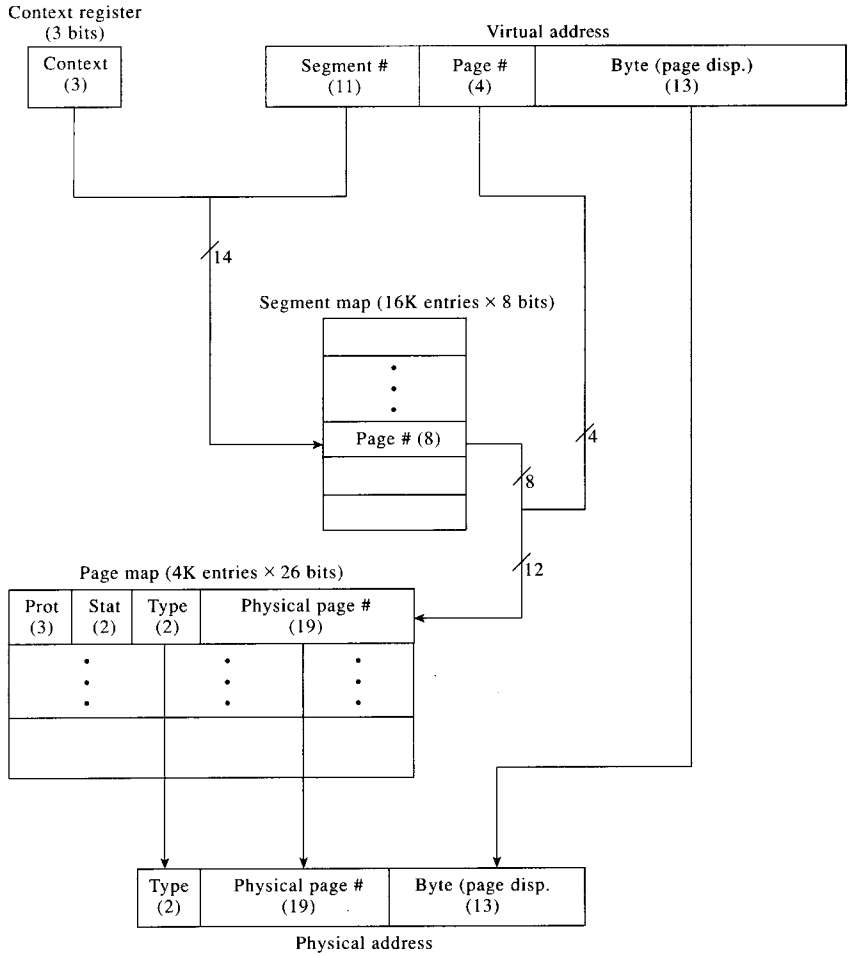



**Figure 8.39** Sun-3 MMU address translation (Courtesy of Sun Microsystems Inc.)

*Supervisor bit* When set, this bit indicates that the page may be accessed only when the processor is in supervisory mode; user mode accesses are not allowed.

*Statistics field* These bits identify the most recently used pages. When a page is successfully accessed, the MMU sets the "accessed" bit for that page. If the access is a write, the "modified" bit for that page is also set. These bits are used during page replacement. Only the "modified" pages are transferred to the backing store during page replacement.

*Type bits* These bits indicate which of the four physical spaces the page resides in. Based on these bits, the memory access request is routed to main memory, the onboard I/O bus, or the VME bus. The memory may reside on the onboard I/O bus or VME bus, and such memory can be mapped into user programs but cannot be used as the main memory.

The MMU overlaps the address translation with physical memory access. When the address translation is started, the 13-bit displacement value is sent to the memory as a row address. This activates all the memory cells in every page that corresponds to that row and sets them to fetch their contents. When the translation is completed, the translated part of the address is sent as a column address to the memory. By this time, the memory cycle is almost complete, and the translated address selcts the memory word at the appropriate row. Due to this overlapped operation, the translation time is effectively zero.

The MMU controls all accesses to the memory including DMA by devices. In the so-called *direct virtual memory access* (DVMA) mode, the devices are given virtual addresses by their drivers. The MMU interposes itself between devices and the memory and translates the virtual addresses to real addresses. This is transparent to devices. This process places a high load on the address translation logic. To avoid the saturation of translation logic in the MMU due to this load, a separate translation pipeline is provided for DMA operations.

Sun-3 systems use two different cache systems. The 3/100 series has no external cache but uses the on-chip cache of the MC68020. The 3/200 series uses an external 64-Kbyte cache (Fig. 8.40) that works closely with the MMU. The cache is organized into 4096 lines of 16 bytes each and is direct mapped. The tag area is in a RAM, and a comparator is used for tag matching. The cache uses the write-back mechanism. Since the cache is maintained in a two-port RAM, the write-back operations can be performed in parallel with the cache access by the processor. The cache works with virtual rather than real addresses. The address translation is started on each memory reference in parallel with the cache search. If there is a cache hit, the translation is aborted. The reader is referred to the manufacturer's manual for further details.

## 8.5 SUMMARY

The memory subsystem of all modern-day computer systems consists of a memory hierarchy ranging from the fast and most expensive processor reg-

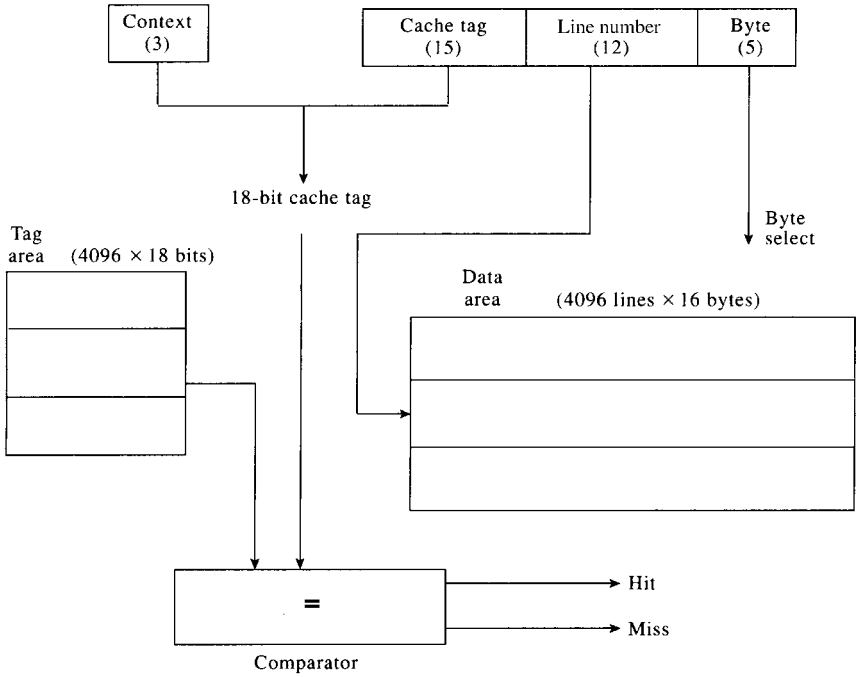



**Figure 8.40** Sun-3 cache (Courtesy of Sun Microsystems Inc.)

isters to the slow and least expensive secondary storage devices. The purpose of the hierarchy is the capacity, speed, and cost tradeoff to achieve the desired capacity and speed at the lowest possible cost. This chapter described the popular schemes for enhancing the speed and capacity of the memory system. The virtual memory schemes are now common even in microcomputer systems. The modern-day microprocessors offer on-chip caches and memory-management hardware to enable building of large and fast memory systems. As the memory capacity increases, the addressing range of the processor becomes a limitation. The popular schemes to extend the addressing range were also described in this chapter.

## REFERENCES

Belady, L. A. "A Study of Replacement Algorithms for a Virtual Storage Computer", *IBM Systems Journal*, Vol. 5, No. 2, 1966, pp. 78–101.

Bell, J. D., D. Casanet, and C. G. Bell. "An Investigation of Alternative Cache Organization", *IEEE Transactions on Computers*, Vol. C-23, April 1974, pp, 346–351.

Cragon, G. C., *Memory Systems and Pipelined Processors*, Boston, MA: Jones and Bartlett, 1996.

Hill, M. D., Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.

*Intel Architecture Software Developer's Manuals – Volume 1: Basic Architecture (24319001.pdf): Volume 2: Instruction Set Reference (24319101.pdf) and Volume 3: System Programming Guide (24319201.pdf)* at http://www.intel.-com/design/PentiumII/manuals, Santa Clara, CA: Intel Corp.

Macgregor, D., D. Mothersole, and B. Moyer. "The Motorola 68020," IEEE Micro, Vol. 4, No. 4, August 1984, pp. 101–118.

MC68851 Paged Memory Management Unit User's Manual. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.

Silberschatz, A. and Galvin, P. B. *Operating System Concepts*, Reading, MA: Addison Wesley, 1997.

Sun-3 Architecture: A Sun Technical Report. Mountain View, Calif.: Sun Microsystems Inc., 1986.

Tannenbaum, A. S. and Goodman, J. R. *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall, 1998.

Tannenbaum, A. S., Woodhull, A. S. and Woodhull, A. *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1997.

*VAX Hardware Handbook*, Maynard, MA: Digital Equipment Corp., 1979.

## PROBLEMS

8.1 Study the memory system characteristics of computer systems you have access to with reference to the features described in this chapter.

8.2 Show the schematic diagram of a 64 KB memory system, using 8 KB memory blocks, assuming (a) interleaving, (b) blocking, and (c) memory system to consist of four subsystems, with interleaving in the subsystem and blocking across the subsystems.

8.3 In each of the designs in Problem 8.2, show the physical location of the following addresses: 0, 48, 356, 8192.

8.4 Assume that ASC memory is built by (two-way) interleaving two 32-Kword blocks. What is the effect of this memory organization on the microinstruction sequences for LDA and TCA instructions? Rewrite these sequences to efficiently utilize the interleaved memory.

8.5 Assume that ASC memory is built using eight interleaved blocks of 8K words each. Include an instruction buffer eight words long into the ASC control unit. Develop the microoperations needed to manage this buffer. Assume that the buffer will be refilled (as a block) when the last instruction is fetched or when a jump is to an address beyond the address range in the buffer.

8.6 Rework Problem 8.5 assuming that the eight memory blocks are banked rather than interleaved.

8.7 Generalize Eqs. (8.2) and (8.3) to $n$-level memory hierarchies.

8.8 The characteristics of a four-level memory hierarchy are shown below:

| Level | Memory type | Access time | Hit ratio |
|---|---|---|---|
| 1 | Cache | 100 ns | 0.8 |
| 2 | Main | 1000 ns | 0.85 |
| 3 | Disk | 1 Ms | 0.9 |
| 4 | Tape | 50 ms | 1.0 |

What is the average access time?

8.9 Given that, in a virtual memory system, if the probability of a page fault is $p$, the main memory access time is $t_m$ and the time required to move a secondary page into the main memory is $t_s$, derive an expression for the average access time $t_a$.

8.10 A computer system has a 64 KB main memory and a 4 KB (data area only) cache. There are 8 bytes/cache line. Determine (1) the number of comparators needed and (2) the size of the tag field, for each of the following mapping schemes:

a. Fully associative
b. Direct
c. Four-way set-associative.

8.11 Show the schematic diagrams of the cache memory in Problem 8.10 assuming that the data and tag areas of the cache are built out of 128 byte RAM ICs.

8.12 In Problem 8.10, assume that the access time of the main memory is 10 times that of the cache, and the cache hit ratio is 0.85. If the access efficiency is defined as the ratio of the average access time with a cache to that without a cache, determine the access efficiency.

8.13 In Problems 8.12, if the cache access time is 100 ns, what hit ratio would be required to achieve an average access time of 500 ns?

8.14 A computer system has 128 KB of secondary storage and an 8 KB main memory. The page size is 512 bytes. Design a virtual memory scheme using (a) direct and (b) fully associative mapping.

8.15 Determine the minimum and maximum page-table sizes and the corresponding number of accesses needed to search the page table, for each scheme in Problem 8.14.

8.16 A memory system has the following characteristics: access times: cache, 100 ns; main memory, 1000 ns; TLB, 40 ns; secondary storage, 3000 ns. The page table is in main memory, and the page-table search requires only one access to the main memory. Determine the minimum and maximum access times for the memory system.

8.17 A processor has a direct addressing range of 64 KB. Show two schematics to extend the address range to 512 KB.

8.18 In a machine with both cache and virtual memories, is the cache tag compared with the virtual address or the physical address? Justify your answer.

8.19 Assume that a system has both main-memory and disk caches. Discuss how write-through and write-back mechanisms work on this machine. What are the advantages and disadvantages of each mechanism?

8.20 What is the difference between bank switching and paging?

# 9
# Control Unit Enhancement

Two popular implementations of the control unit (HCU and MCU) were described in Chapter 5. Chapters 6, 7, and 8 provided the details of architectural features found in modern-day machines. Enhancement of ASC to include any of these features would require a modification to the control unit. This chapter describes the features found in practical machines that enhance the performance of the control unit. The following parameters are usually considered in the design of a control unit:

*Speed*. The control unit should generate control signals fast enough to utilize the processor bus structure most efficiently and minimize the instruction execution time.

*Cost and complexity*. The control unit is the most complex subsystem of any processing system. The complexity should be reduced as much as possible to make the maintenance easier and the cost low. In general, random logic implementations of the control unit (i.e. HCU) tend to be the most complex, while ROM-based designs (i.e. MCU) tend to be the least complex.

*Flexibility*. HCUs are inflexible in terms of adding new architectural features to the processor since they require a redesign of the hardware. MCUs offer a very high flexibility since microprograms can be easily updated without a substantial redesign of the hardware involved.

With the advances in hardware technology, faster and more versatile processors are introduced to the market very rapidly. This requires that the design cycle time for newer processors must be as small as possible. Since the design costs must be recovered over a short life span of the new processor, they must be minimized. MCUs offer such flexibility and low-cost redesign capabilities, although they are inherently slow compared to HCUs. This speed differential between the two designs is getting smaller, since in the

current technology, the MCU is fabricated on the same IC (i.e. with the same technology) as the rest of the processor. We will concentrate on the popular speed-enhancement techniques used in contemporary machines in this chapter.

## 9.1 SPEED ENHANCEMENT

In ASC, the control unit fetches an instruction, decodes it and executes it, before fetching the next instruction as dictated by the program logic. That is, the control unit brings about the instruction cycles one after the other. With this *serial* instruction execution mode, the only way to increase the speed of execution of the overall program is to minimize the instruction cycle time of the individual instructions. This concept is called the *instruction cycle speedup*. The program execution time can be reduced further, if the instruction cycles can be overlapped. That is, if the next instruction can be fetched and/or decoded during the current instruction cycle. This overlapped operation mode is termed *instruction execution overlap* or more commonly *pipelining*. Another obvious technique would be to bring about the instruction cycles of more than one instruction simultaneously. This is the *parallel* mode of instruction execution. We will now describe these speed enhancement mechanisms.

### 9.1.1 Instruction Cycle Speedup

Recall that the *processor cycle time* (i.e., minor cycle time) depends on the register transfer time on the processor bus structure. If the processor structure consists of multiple buses, it is possible to perform several register transfers simultaneously. This requires that the control unit produce the appropriate control signals simultaneously.

In a *synchronous* HCU, the processor cycle time is fixed by the slowest register transfer. Thus even the fastest register transfer operation consumes a complete processor cycle. In an *asynchronous* HCU, the completion of one register transfer triggers the next; therefore, if properly designed, the asynchronous HCU would be faster than the synchronous HCU. Since the design and maintenance of an asynchronous HCU is difficult, the majority of the practical processors have synchronous HCUs. An MCU is slower than an HCU since the microinstruction execution time is the sum of processor cycle time and the CROM access time.

The HCU of ASC has the simplest configuration possible. Each instruction cycle is divided into one or more phases (states or *major cycles*), each phase consisting of four processor cycles (i.e., *minor cycles*). A majority

of actual control units are synchronous control units that are, in essence, enhanced versions of the ASC control unit. For example, it is not necessary to use up a complete major cycle if the microoperations corresponding to an instruction execution (or fetch or defer) can be completed in a part of the major cycle. The only optimization performed in the ASC control unit was to reduce the number of major cycles needed to execute certain instructions (SHR, SHL) by not entering an execute cycle, since all the required microoperations to implement those instructions could be completed in one major cycle. Further optimization is possible. For example, the microoperations corresponding to the execution of each branch instruction (BRU, BIP, BIN) could all be completed in one minor cycle rather than in a complete major cycle as they are in the ASC control unit. Thus, three minor cycles could be saved in the execution of branch instructions by returning to fetch cycle after the first minor cycle in the execute cycle. When such enhancements are implemented, the state-change circuitry of the control unit becomes more complex but the execution speed increases.

Note that in the case of an MCU, the concept of the major cycle is not present, and the basic unit we work with is the minor cycle (i.e., processor cycle time + CROM access time). Thus the lengths of microprograms corresponding to each instruction are different. Each microinstruction is executed in one minor cycle, and the microprograms do not include any idle cycles. In developing the microprogram for ASC (Table 5.6) the microoperation sequences from the HCU design were reorganized to make them as short as possible.

Section 9.5 provides the instruction cycle details of Intel 8080, to illustrate the instruction cycle speedup concept. Although this is an obsolete processor, it was selected for its simplicity. The more recent Intel processors adopt these techniques very extensively.

### 9.1.2 Instruction Execution Overlap

Note that in INTEL 8080, for instructions such as ADD *r*, once the memory operand is fetched into the CPU, addition is performed while the CPU is fetching the next instruction in sequence from the memory. This overlap of instruction fetch and execute phases increases the program execution speed.

In general, the control unit can be envisioned as a device with three subfunctions: fetch, decode (or address computation), and execute. If the control unit is designed in a modular form with one module for each of these functions, it is possible to overlap the instruction-processing functions.

The overlapped processing is brought about by a *pipeline*. A *pipeline* is a structure that, like an automobile assembly line, consists of several stations, each of which is capable of performing a certain subtask. The work

flows from one station to the next. As the work leaves a station, the subsequent unit of the work is picked up by that station. When the work leaves the last station in the pipeline the task is complete. If the pipeline has $N$ stations and the work stays at each station for $T$ seconds, the complete processing time for a task is $(N \times T)$ seconds. But since all the $N$ stations are working in an overlapped manner (on various tasks), the pipeline outputs one completed task every $T$ seconds (after an initial period in which the pipeline is being filled).

Figure 9.1 introduces the concept of an instruction processing *pipeline*. The control unit has three modules. The processing sequence is shown in (b). Any time after $t_2$, the first module will be fetching instruction (I + 1), the second module will be decoding instruction I, while the last module will be executing instruction (I – 1). From $t_3$ onwards, the pipeline flows full and the throughput is one instruction per time slot.

For simplicity, we have assumed that each module in the above pipeline consumes the same amount of processing time. If such equal time partitioning of the processing task cannot be made, intermediate registers to hold the results and flags that indicate the completion of one task and beginning of the next task are needed.

We have assumed that the instructions are always executed in the sequence they appear in the program. This assumption is valid as long as the program does not contain a branch instruction. When a branch instruc-

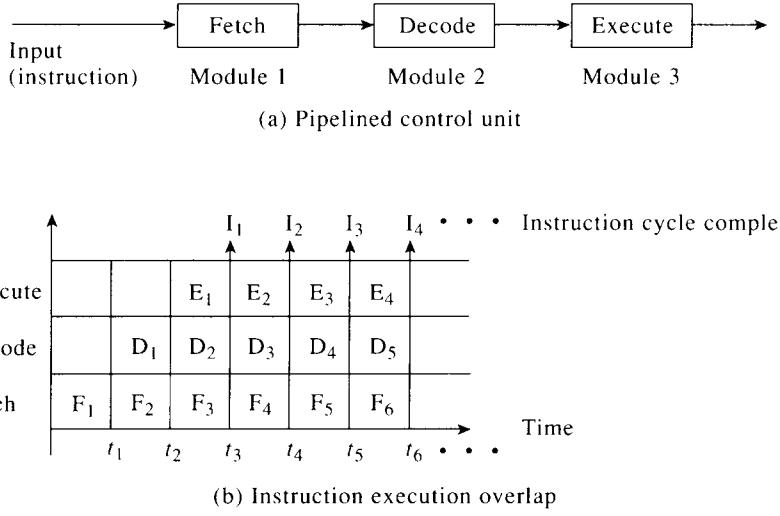



**Figure 9.1** Pipelined instruction processing

tion is encountered, the next instruction is to be fetched from the target address of the branch instruction. If it is a conditional branch, the target address would not be known until the instruction reaches the execute stage. If a branch is indeed taken, then the instructions following the branch instruction that are already in the pipeline need to be discarded, and the pipeline needs to be filled from the target address. Another approach would be to stop fetching subsequent instructions into the pipeline, once the branch instruction is fetched, until the target address is known. The former approach is preferred for handling conditional branches since there is a good chance that the branch might not occur; in that case, the pipeline would flow full. There is an inefficiency in the pipeline flow only when a branch occurs. For unconditional branches, the latter approach can be used. The following mechanisms have been used to handle the conditional branch inefficiency of the pipeline structures.

### Branch Prediction

It is sometimes possible to predict the target address of a conditional branch based on the execution characteristics. For example, the target address of a branch instruction controlling the iterations through a loop is most likely the address of the first instruction in the loop, except for the last time through the iteration. If this compiler-dependent characteristic is known, the most likely target address can be used to fetch the subsequent instructions into the pipeline.

### Branch History

If the statistics on the previous branches for each instruction are maintained, the most likely target address can be inferred.

### Delayed Branching

In the three-stage pipeline of Fig. 9.1, two instructions following the branch instruction will have entered the pipeline by the time the target address is determined. If, in the instruction stream, the branch instruction is moved ahead two positions from where it logically appears in the program, the execution of instructions in the pipeline can proceed while the target address is determined.

All three stages of the instruction processing pipeline can potentially access the main memory simultaneously, thereby increasing the traffic on

the memory bus. A memory system should provide for such simultaneous access and high throughput by multiport, interleaving, and banking schemes.

In order to implement the control unit as a pipeline, each stage should be designed to operate independently, performing its own function while sharing resources such as the processor bus structure and the main memory system. Such designs become very complex. Refer to the books listed in the Reference section for further details.

The pipeline concept is now used very extensively in all modern processors. It is typical for the processors today to have four or five stages in their pipelines. As the hardware technology progresses, processors with deeper pipelines (i.e., pipelines with larger numbers of stages) have been introduced. These processors belong to the so-called *superpipelined* processor class. Section 9.5 and subsequent chapters provide some examples.

### 9.1.3 Parallel Instruction Execution

As seen by the description in Chapter 7, the Intel Pentium series of processors contain two execution units: the integer unit and the floating-point unit. The control unit of these processors fetches instructions from the same instruction stream (i.e., program), decodes them and delivers them to the appropriate execution unit. Thus, the execution units would be working in parallel, each executing its own instruction. The control unit must now be capable of creating these parallel streams of execution and synchronizing the two streams appropriately, based on the precedence constraints imposed by the program. That is, the result of the computation must be the same, whether the program is executed by serial or parallel execution of instructions. This class of architectures, where more than one instruction stream is processed simultaneously, is called a *superscalar* architecture. Section 9.5 and subsequent chapters provide some examples.

### 9.1.4 Instruction Buffer and Cache

The instruction buffer schemes used by processors such as INTEL 8086 and CDC 6600, and the instruction cache schemes used by processors such as MC68020 also bring about instruction processing overlap, although at the complete instruction level. That is, the operation of fetching instructions into the buffer is overlapped with the operation of retrieving and executing instructions that are in the buffer.

## 9.2 HARDWIRED CONTROL UNITS

All the speedup techniques described in the previous section have been adopted by HCUs of practical machines. As mentioned earlier, the main advantage of the HCU is its speed, while the disadvantage is its inflexibility. Although asynchronous HCUs offer a higher speed capability than synchronous HCUs, the majority of practical machines have synchronous HCUs because they have the simpler design of the two.

In the current VLSI era, complete processing systems are fabricated on a single IC. Because the control unit is the most complex unit, it occupies a large percentage of the chip "real estate." Its complexity increases as the number of instructions in the instruction set of the machine increases. Studies have shown that, in practice, only a few instructions are used by programmers, although the machine offers a large instruction set. One way, then, of reducing the complexity of the control unit is to have a small instruction set. Such machines with small sets of powerful instructions are called *reduced instruction set computers* (RISC). Section 9.4 describes RISCs further.

## 9.3 MICROPROGRAMMED CONTROL UNITS

The execution time for an instruction is proportional to the number of microoperations required and hence the length of microprogram sequence for the instruction. Since a microprogrammed control unit (MCU) starts fetching the next instruction once the last microoperation of the current instruction microprogram is executed, MCU can be treated as an asynchronous control unit. An MCU is slower than the hardwired control unit because of the addition of CROM access time to the register transfer time. But it is more flexible than hardwired CU and requires minimum changes in the hardware if the instruction set is to be modified or enhanced.

All the speedup techniques described in Section 9.1 are used in practical MCUs. Another level of pipelining, shown in Fig. 9.2, is possible in an MCU. Here, the fetching of the next microinstruction is overlapped with the execution of the current microinstruction.

The CROM word size is one of the design parameters of an MCU. Although the price of ROMs is decreasing, the cost of data path circuits required within the control unit increases as the CROM word size increases. Therefore, the word size should be reduced to reduce the cost of the MCU. We will now examine the microinstruction formats used by practical machines with respect to their cost effectiveness.

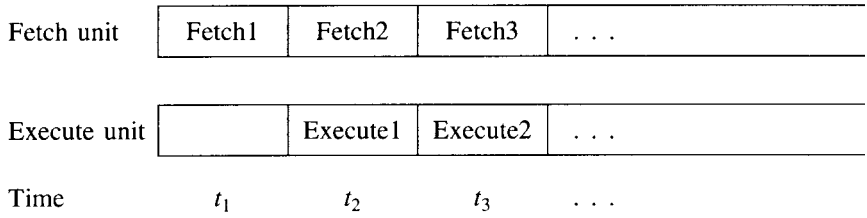



**Figure 9.2** Pipelining in an MCU

The most common format for a microinstruction is shown in Fig. 9.3(a). The "instruction" portion of the microinstruction is used in generating the control signals, and the "address" portion indicates the address of the next microinstruction. Execution of such a microinstruction corresponds to the generation of control signals and transferring the address portion to the $\mu$MAR to retrieve the next microinstruction. The advantage of this format is that very little external circuitry is needed to generate the next microinstruction address, while the disadvantage is that conditional branches in the microprogram cannot be easily coded. The format shown in (b) allows for conditional branching. It is now assumed that when the condition is not satisfied, the $\mu$MAR is simply incremented to point to the next microinstruction in sequence. However, this requires additional

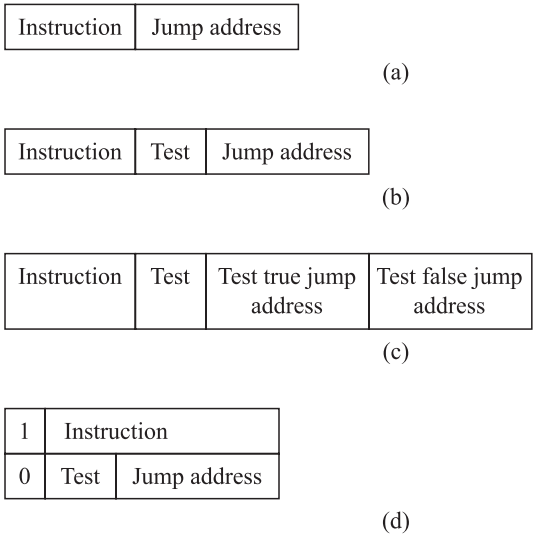
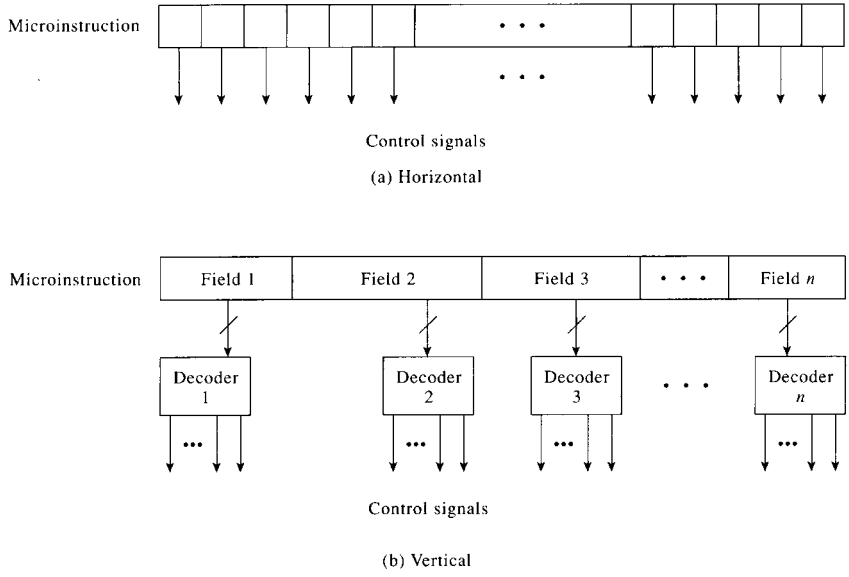



**Figure 9.3** Examples of microinstruction formats

$\mu$MAR circuitry. The microinstruction format shown in (c) explicitly codes the jump addresses corresponding to both the outcomes of the test condition, thus reducing the external $\mu$MAR circuitry. The CROM word size will be large in all the above formats since they contain one or more address fields. It is possible to reduce the CROM word size if the address representation can be made implicit. The format shown in (d) is similar to the one used by the MCU of ASC. This format distinguishes between the two types of microinstructions through the $\mu$opcode: Type 1 microinstruction produces control signals, while type 0 manages the microprogram flow. Incrementing of $\mu$MAR after the execution of a type 1 microinstruction is implicit in this format. As seen in Chapter 5, this format of microinstruction requires fairly complex circuitry to manage the $\mu$MAR.

Figure 9.4 shows the two popular forms of encoding the "instruction" portion of a microinstruction. In the *horizontal* (or *unpacked*) microinstruction, each bit in the instruction portion represents a control signal. Hence, all the control signals can be generated simultaneously and no external decoding is needed to generate the control signals, thus making




Note: only the instruction field of the microinstruction is shown; the address and test fields are not shown.

**Figure 9.4** Popular microinstruction-encoding formats

the MCU fast. The disadvantage is that this requires larger CROM words. Also, instruction encoding is cumbersome because a thorough familiarity with the processor hardware structure is needed to prevent the generation of control signals that cause conflicting operations in the processor hardware.

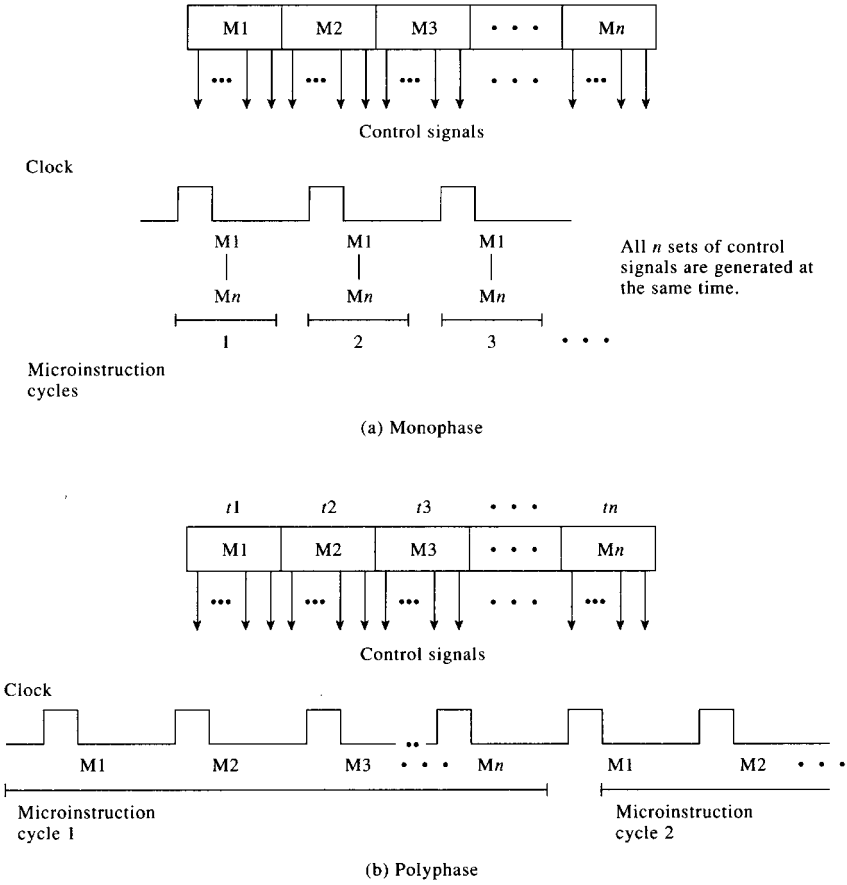
In the *vertical* (or *packed*) microinstruction, the instruction is divided into several fields, each field corresponding to either a resource or a function in the processing hardware. (In the design of ASC MCU, each field corresponded to a resource such as ALU, BUS1, etc.). Vertical microinstruction reduces the CROM word size; but the decoders needed to generate control signals from each field of the instruction contribute to the delays in control signals. Encoding for vertical microinstruction is easier than that for horizontal microinstruction because of the former's function/resource partitioning.

In the foregoing discussion, we have assumed that all the control signals implied by the microinstruction are generated simultaneously and the next clock pulse fetches a new microinstruction. This type of microinstruction encoding is called *monophase encoding*. It is also possible to associate each field or each bit in the microinstruction with a time value. That is, the execution of each microinstruction now requires more than one clock pulse. This type of microinstruction encoding is called *polyphase encoding*. Figure 9.5(a) shows the monophase encoding, where all the control signals are generated simultaneously at the clock pulse. Figure 9.5(b) shows an *n*-phase encoding where microoperations M1 through M*n* are associated with time values *t*1 through *tn* respectively.

## 9.4 REDUCED INSTRUCTION SET COMPUTERS (RISC)

The advent of VLSI provided the capability to fabricate a complete processor on an IC chip. In such ICs, the control unit typically occupied a large portion (of the order of 50–60%) of the chip area, thereby restricting the number of processor functions that could be implemented in hardware. A solution to this problem was the design of processors with simpler control units. RISC architectures enable the use of a simple control unit since their instruction sets tend to be small. Simplification of the control unit was the main aim of the RISC designs first initiated at IBM in 1975 and later, in 1980, at the University of California, Berkeley. The characteristics and the definition of RISC have changed considerably since those early designs. There is currently no single accepted definition of RISC.

One of the motivations for the creation of RISC was the criticism of certain problems inherent in the design of the established complex instruction set computers (CISC). A CISC has a relatively large number of com-




Notes:
1. M1–M*n* are the *n* sets of control signals implied by the microinstruction.
2. *t*1–*tn* are microcontrol unit clock pulses (phases).

**Figure 9.5** Phase encoding of microinstructions

plicated, time-consuming instructions. It also has a relatively large number of addressing modes and different instruction formats. These in turn result in the necessity of having a complex control unit to decode and execute these instructions. Some of the instructions may be so complex that they are not necessarily faster than a sequence of several RISC instructions that could replace them and perform the same function.

In order to provide a perspective, let us look at an example of a computing system that could be considered the opposite of a RISC – a

complex instruction set computer (CISC). One such system is the DEC VAX-11/780 with its 304 instructions and 16 addressing modes. It has 16 32-bit registers. The VAX supports a considerable number of data types, among which are 6 types of integers, 4 types of floating points, packed decimal strings, character strings, variable-length bit fields, numeric strings, and more. An instruction can be as short as 2 bytes and as long as 14 bytes. There may be up to 6 operand specifiers. As another example of a CISC-type system, we can even take a microprocessor, such as the 32-bit Motorola MC68020. It also has 16 general-purpose CPU registers, recognizes 7 data types and implements 18 addressing modes. The 68020 instructions can be from one word (16 bits) to 11 words in length.

All in all, the MC68020 as well as the VAX-11/780 (and many other CISCs) have a great variety of data and instruction formats, addressing modes, and instructions. A direct consequence of this variety is the highly complex control unit. For instance, even in the less sophisticated MC68000, the control unit takes up over 60% of the chip area.

Naturally, a RISC-type system is expected to have fewer than 304 instructions. There is no exact consensus about the number of instructions that a RISC system should have. The Berkeley RISC I had 31, the Stanford MIPS had over 60, and the IBM 801 had over 100. Although the RISC design is supposed to minimize the instruction count, that in itself is not the definitive characteristic of a RISC. The instruction environment is also simplified, by reducing the number of addressing modes, by reducing the number of instruction formats, and simplifying the design of the control unit. Based on the RISC designs reported so far, we can look at the following list of RISC attributes as an informal definition of a RISC.

1. A relatively low number of instructions (preferably less than 100).
2. A low number of addressing modes (preferably one or two).
3. A low number of instruction formats (preferably one or two).
4. Single-cycle execution of all instructions.
5. Memory access performed by load/store instructions only.
6. The CPU has a relatively large register set (over 32; there are 138 registers in RISC II). Ideally, most operations can be done register-to-register. Memory access operations are minimized.
7. A hardwired control unit (may be changed to microprogrammed as the technology develops).
8. An effort is made to support high-level language (HLL) operations inherently in the machine design by using a judicious choice of instructions and optimized (with respect to the large CPU register set) compilers.

It should be stressed that the above points are to be considered as a flexible framework for a definition of a RISC machine, rather than a list of design attributes common to most of the RISC systems. The boundaries between RISC and CISC have not been rigidly fixed. Still, the above points give at least an idea of what to expect in a RISC system.

RISC machines are used for many high-performance applications. Embedded controllers are a good example. They can also be used as building blocks for more complex multiprocessing systems. It is precisely its speed and simplicity that make a RISC appropriate for such applications.

The complexity of the CISC control unit hardware implies a longer design time. It also increases the probability of a larger number of design errors. And those errors are subsequently difficult, time consuming, and costly to locate and correct.

A large instruction set presents too large a selection of choices for the compiler of any high-level language (HLL). This in turn makes it more difficult to design the optimizing stage of a CISC compiler. This stage would have to be longer and more complicated in a CISC machine. Furthermore, the results of this "optimization" may not always yield the most efficient and fastest machine language code.

Some CISC instruction sets contain a number of instructions that are particularly specialized to fit certain HLL instructions. However, a machine language instruction that fits one HLL may be redundant for another and would require excessive effort on the part of the designer. Such a machine may have a relatively low cost–benefit factor.

A RISC has relatively few instructions, few addressing modes, and a few instruction formats. As a result, a relatively small and simple (compared to CISC) decode and execute hardware subsystem of the control unit is required. The chip area of the control unit is considerably reduced. For example: The control area on RISC I constitutes 6% of the chip area; on RISC II, 10%; and on the Motorola MC68020, 68%. In general, the control area for CISCs takes over 50% of the chip area. Therefore, on a RISC VLSI chip, there is more area available for other features. As a result of the considerable reduction of the control area, the RISC designer can fit a large number of CPU registers on the chip. This in turn enhances the throughput and the HLL support.

The control unit of a RISC is simpler, occupies a smaller chip area and provides faster execution of instructions. A small instruction set and a small number of addressing modes and instruction formats imply a faster decoding process. A large number of CPU registers permit programming that reduces memory accesses. Since CPU register-to-register operations are much faster than memory accesses, the overall speed is increased. Since all instructions have the same length, and all execute in one cycle, we obtain a

streamlined instruction handling that is particularly well-suited to pipelined implementation.

Since the total number of instructions in a RISC system is small, a compiler (for any HLL), while attempting to realize a certain operation in machine language, usually has only a single choice, as opposed to a possibility of several choices in a CISC. This makes that part of the compiler shorter and simpler for a RISC.

The availability of a relatively large number of CPU registers in a RISC permits a more efficient code optimization stage in a compiler by maximizing the number of faster register-to-register operations and minimizing the number of slower memory accesses.

All in all, a RISC instruction set presents a reduced burden on the compiler writer. This tends to reduce the time of design of RISC compilers and their costs. Since a RISC has a small number of instructions, a number of functions performed on CISCs by a single instruction need two, three, or more instructions on a RISC. This causes the RISC code to be longer and constitutes an extra burden on the machine and assembly language programmer. The longer RISC programs consequently require more memory locations for their storage.

It is argued that RISC performance is due primarily to extensive register sets and not to the RISC principle in general. Again, from the VLSI standpoint, it is precisely due to the RISC principles, which permitted a significant reduction of the control area, that the RISC designers were able to fit so many CPU registers on a chip in the first place.

It has been argued that RISCs may not be efficient with respect to operating systems and other auxiliary operations. The efficiency of RISCs with respect to certain compilers and bench marks has already been established. There is no substantiated reason to doubt that efficient operating systems and utility routines can be generated for a RISC.

A number of RISC processors have been introduced with various performance ratings, register and bus structures, and instruction-set characteristics. The reader is referred to the magazines listed in the References section for further details.

## 9.5 EXAMPLE SYSTEMS

Most of the concepts introduced in this chapter were used by the processors described in earlier chapters of this book. In this section we provide additional examples. The first example (Intel 8080) is selected for its simplicity, although it is now obsolete. The Digital Equipment Corporation's (DEC) VAX is still a popular minicomputer architecture, although DEC has been

acquired by Compaq Computer Corporation. Motorola 88000 series of processors are selected to represent early RISC architectures. Finally, the MIPS R10000 processor from Silicon Graphics Inc. represents a contemporary RISC architecture.

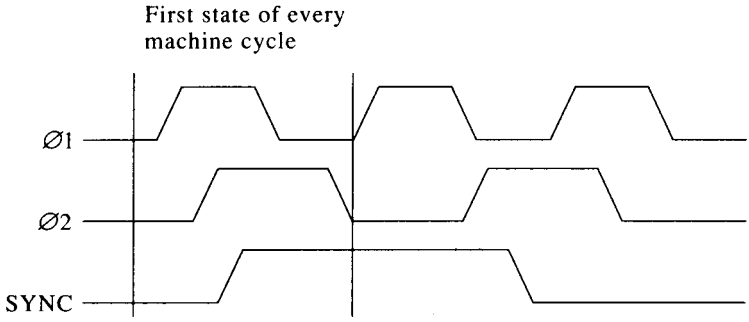
### 9.5.1 Intel 8080

The INTEL 8080 has a synchronous HCU operating with a major cycle (or *machine cycle* in Intel terminology) consisting of three, four, or five minor cycles (or *states* in Intel terminology). An instruction cycle consists of one to five major cycles, depending on the type of instruction. The machine cycles are controlled by a two-phase nonoverlapping clock. "SYNC" identifies the beginning of a machine cycle. There are ten types of machine cycles: fetch, memory read, memory write, stack read, stack write, input, output, interrupt acknowledge, halt acknowledge, and interrupt acknowledge while halt.

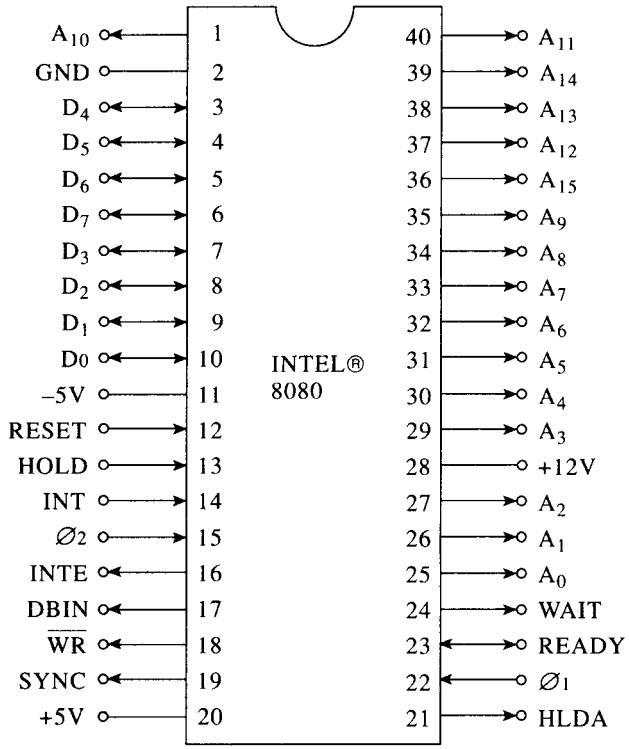
Instructions in the 8080 contain 1 to 3 bytes. Each instruction requires from one to five machine or memory cycles for fetching and execution. Machine cycles are called M1, M2, M3, M4, and M5. Each machine cycle requires from three to five states, T1, T2, T3, T4, and T5, for its completion. Each state has the duration of one clock period (0.5 microseconds). There are three other states (WAIT, HOLD, and HALT), which last from one to an indefinite number of clock periods, as controlled by external signals. Machine cycle M1 is always operationcode fetch cycle and lasts four or five clock periods. Machine cycles M2, M3, M4, and M5 normally last three clock periods each.

To further understand the basic timing operation of the INTEL 8080, refer to the instruction cycle shown in Fig. 9.6. During T1, the content of the program counter is sent to the address bus, SYNC is true, and the data bus contains the status information pertaining to the cycle that is currently being initiated. T1 is always followed by another state, T2, during which the condition of the READ, HOLD, and HALT acknowledge signals are tested. If READY is true, T3 can be entered; otherwise, the CPU will go into the WAIT state ($T_w$) and stays there for as long as READY is false. READY thus allows the CPU speed to be synchronized to a memory with any access time or to any input device. The user, by properly controlling the READY line, can single-step through a program.

During T3, the data coming from memory are available on the data bus and are transferred into the instruction register (during M1 only). The instruction decoder and control sections then generate the basic signals to control the internal data transfer, the timing, and the machine cycle requirements of the instruction.



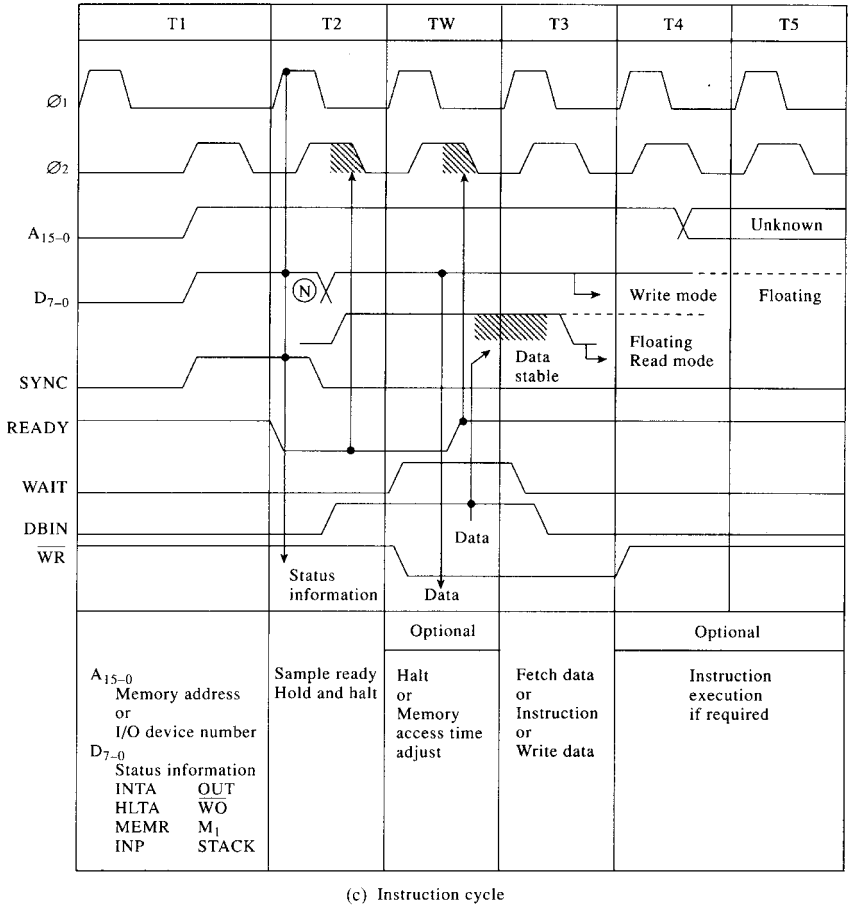

(a) Nonoverlapping clock




(b) Pinout diagram

(*continues*)

**Figure 9.6** INTEL 8080 (Reprinted by permission of Intel Corporation. Copyright 1981. All mnemonics copyright Intel Corporation 1981.)




(c) Instruction cycle

**Figure 9.6** (Continued)

At the end of T4 (if the cycle is complete) or at the end of T5 (if it is not), the 8080 goes back to T1 and enters machine cycle M2, unless the instruction required only one machine cycle for its execution. In such cases, a new M1 cycle is entered. The loop is repeated for as many cycles and states as may be required by the instruction.

Instruction-state requirements range from a minimum of four states for non-memory referencing instructions (such a register and accumulator arithmetic instructions) and up to eighteen states for the most complex instructions (such as instructions to exchange the contents of

registers $H$ and $L$ with the contents of the two top locations of the stack). At the maximum clock frequency of 2 MHz, this means that all instructions will be executed in intervals from 2 to 9 $\mu$s. If a HALT instruction is executed, the processor enters a WAIT state and remains there until an interrupt is received.

Figure 9.7 shows the microoperation sequences of two instructions (one column for each instruction). The first three states (T1, T2, and T3) of the fetch machine cycle (M1) are the same for all instructions.

### 9.5.2 Digital Equipment Corporation's (DEC) VAX-11/750

Figures 9.8 and 9.9 depict Digital Equipment Corporation's VAX-11/750 microarchitecture and microinstruction fields. There are three internal buses in the DEC VAX-11/750: the WBUS, MBUS and RBUS. The output of the ALU, unless inhibited, goes on the WBUS. Data to be written into memory and data to be written into the scratch pad registers are taken from the WBUS. States and control information is passed to or from the particular registers via the WBUS.

The MBUS and RBUS provide a source for the super-rotator and the ALU. Data on the MBUS are primarily taken from the main-memory interface registers and from the M scratch pad registers. Data on the RBUS are from the R scratch pad registers and the long literal register. The ALU can perform 2s complement arithmetic, BCD arithmetic, and logical operations. The output of the ALU can be shifted or rotated. There are 56 scratch pad registers. The super-rotator can barrel-shift a 64-bit data element, extract a desired field from a given piece of data, and construct a 32-bit data element according to a variety of specifications. This provides VAX-11/750 with a very efficient bit-manipulation capability. Different components or elements are controlled by the bits in microinstruction fields.

The control store is a 16K, 80-bit word memory. Of this, the low-order 6K words are required to emulate VAX and the next 2K words are required for the remote diagnostic module (RDM). This leaves 8K 80-bit words of address space for user control store (UCS) or writable control store of which 1K is actually available as part of the control store.

The VAX Series uses the microarchitecture primarily to emulate the exceptions and interrupt mechanism. Emulation starts with branch on microtest (BUT) bits in the microinstruction.

Some machines are designed with a *writable control store*. The sytem programmers can load a microprogram into the control store (RAM or

(a)

| Mnemonic | | JMP adrs | CALL adrs |
|---|---|---|---|
| Opcode ($D_7 - D_0$) | | 1100 0011 | 1100 1101 |
| M1[1] | T4[3] | X | SP ← SP − 1 |
| | T5 | | |
| M2 | T1 | PC OUT, STATUS[5] | PC OUT, STATUS[5] |
| | T2[2]<br>T3 | PC ← PC + 1<br>Z ← B2[4] | PC ← PC + 1<br>Z ← B2 |
| M3 | T1 | PC OUT, STATUS[5] | PC OUT, STATUS[5] |
| | T2[2]<br>T3 | PC ← PC + 1<br>W ← B3[4] | PC ← PC + 1<br>W ← B3 |
| M4 | T1 | | SP OUT, STATUS[9] |
| | T2[2]<br>T3 | | Data bus ← PCH<br>SP ← SP − 1 |
| M5 | T1 | | SP OUT, STATUS[9] |
| | T2[2]<br>T3 | | Data bus ← PCL |
| | T4[3]<br>T5 | | |
| During the fetch of next instruction | | WZ OUT, STATUS[8]<br>PC ← WZ + 1 | WZ OUT, STATUS[8]<br>PC ← WZ + 1 |

**Figure 9.7** INTEL 8080 microinstructions. (Reprinted by permission of Intel Corporation, copyright 1981. All mnemonics copyright Intel Corporation 1981.)

ROM) on the machine to emulate a target machine or to implement a new instruction on the host. Machines such as the Nanodata QM-1 are called "soft machines," since they do not have an instruction set of their own, but an instruction set required for an application can be devised by microprogramming these machines.

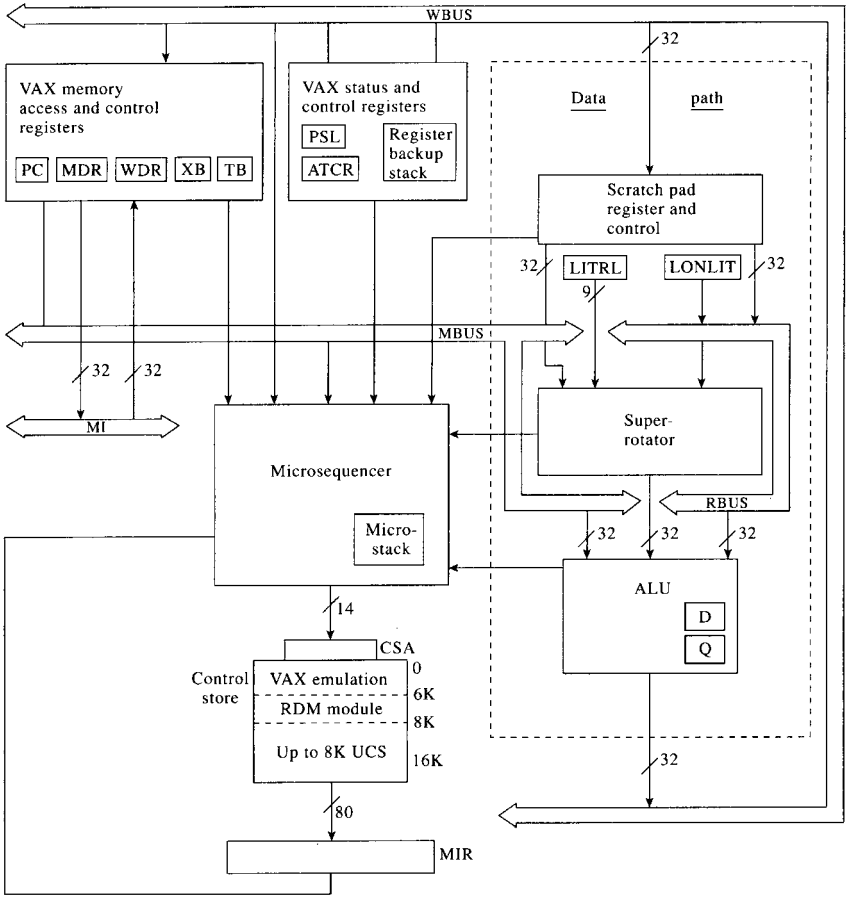



**Figure 9.8** DEC VAX-11/750 microarchitecture. (Copyright, Digital Equipment Corporation, 1979. All Rights Reserved.)

### 9.5.3 Motorola MC88100/88200

The MC88100 is the first processor in the MC8800 family of RISC processors. The MC88000 family also includes the MC88200, a high-speed memory-caching and demand-paged memory management unit. Together they offer the user a powerful RISC system that is claimed to become the new standard in high-performance computing.

1. The MC88100 instruction set contains 51 instructions. These instructions include:

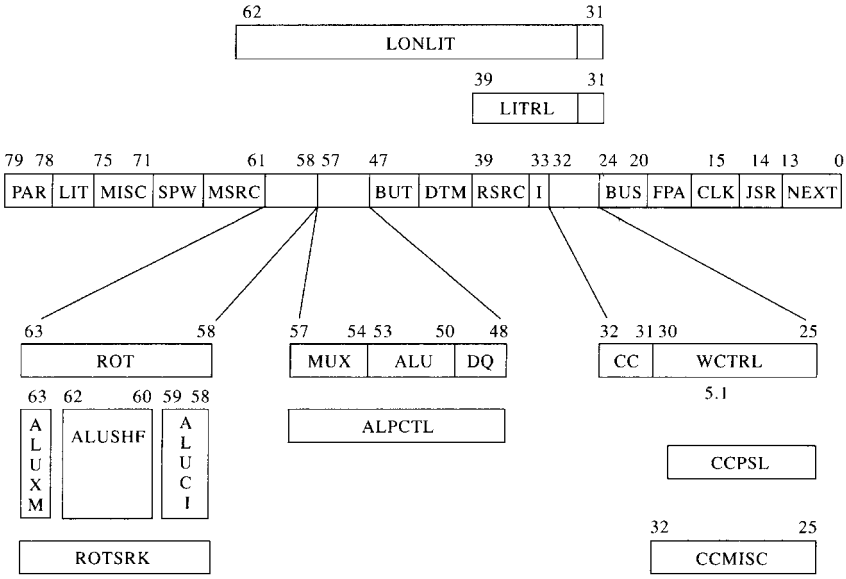



**Figure 9.9** DEC VAX-11/750 microinstruction fields. (Copyright, Digital Equipment Corporation, 1979. All Rights Reserved.)

a. integer add, subtract, multiply, and divide
b. floating-point add, subtract, multiply, and divide
c. logical AND, OR, and XOR
d. bit-field instructions
e. memory-manipulation instructions
f. flow-control instructions.

2. There are a small number of addressing modes: three addressing modes for data memory, four addressing modes for instruction memory, plus three register addressing modes.
3. A fixed-length instruction format is implemented. All instructions are 32 bits long.
4. All instructions are either executed in one processor cycle or dispatched to pipelined execution units that produce results every processor cycle.
5. Memory access is performed by LOAD/STORE instructions.
6. The CPU contains 32 32-bit user reigsters as well as numerous registers used to control pipelines and save the context of the processor during exception processing.
7. The control unit is hardwired.

8. High-level language support exists in the form of procedure parameter registers and a large register set in general.

The most important design feature that contributes to the single-cycle execution of all instructions is the multiple-execution units (see Fig. 9.10): the instruction unit, the data unit, the floating-point unit, and the integer unit. The integer unit and the floating-point unit execute all data manipulation instructions. Data memory accesses are performed by the data unit, and instruction fetches are performed by the instruction unit. They operate both independently and concurrently. These execution units allow the MC88100 to perform up to five operations in parallel. It can access program memory, execute an arithmetic, logical, or bit-field instruction, access data memory, execute a floating-point or integer-divide instruction, or execute a floating-point or integer-multiply instruction.

Three of the execution units are pipelined: the instruction unit, the data unit, and the floating-point unit. The floating-point unit actually has two pipelines, the add pipe and the multiply pipe.

Data interlocks are avoided within the pipelines by using a scoreboard register. Each time an instruction enters the pipe, the bit corresponding to the destination register is set in the scoreboard register. The subsequent instruction entering the pipe checks to see if the bit corresponding to its source registers is set. If so, it waits. Upon completion of the instruction, the pipeline mechanism clears the destination register's bit, freeing it to be used as a source operand.

The MC88100 incorporates delayed branching to reduce pipeline penalties associated with changes in program flow due to conditional branch instructions. This technique enables the instruction fetched after the branch instruction to be optionally executed. Therefore, the pipeline flow is not broken.

There are also three internal buses within the MC88100, two source operand buses and a destination bus. These buses enable the pipeline to access operand registers concurrently. Two different pipes can be accessing the same operand register on one of two source buses.

Data and instructions are accessed via two nonmultiplexed address buses. This scheme, known as the *Harvard architecture*, eliminates bus contention between data accesses and instruction fetches by using the MC88200 (see Fig. 9.11).

The MC88200 contains a memory-management unit as well as a data/instruction cache. The memory-management unit contains two address-translation caches, the BATC and the PATC. The BATC is generally used for high-hit addresses. It is fully associative. The BATC contains ten entries for 512 KB blocks. Eight of these entries are controlled by software, and two

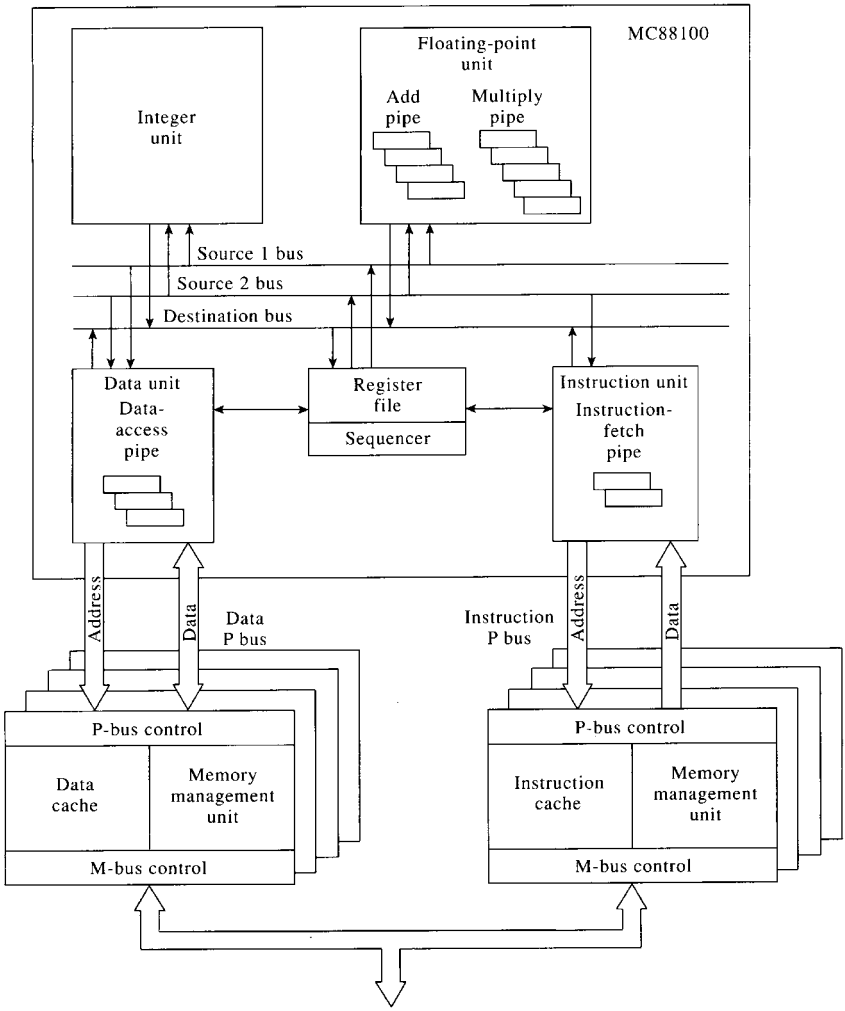



**Figure 9.10** MC88100/MC88200 block diagram. (Courtesy of Motorola Corporation.)

are hardwired to 1 MB of the I/O page. Memory protection is implemented by protection flags. The PATC is generally used for all other accesses. It is also fully associative. The PATC contains 56 entries for 4 KB pages. It is hardware controlled. Here again, memory protection is implemented via protection flags. The address translation tables are standard two-level mapping tables. The MC88200 searches the BATC and PATC in parallel.

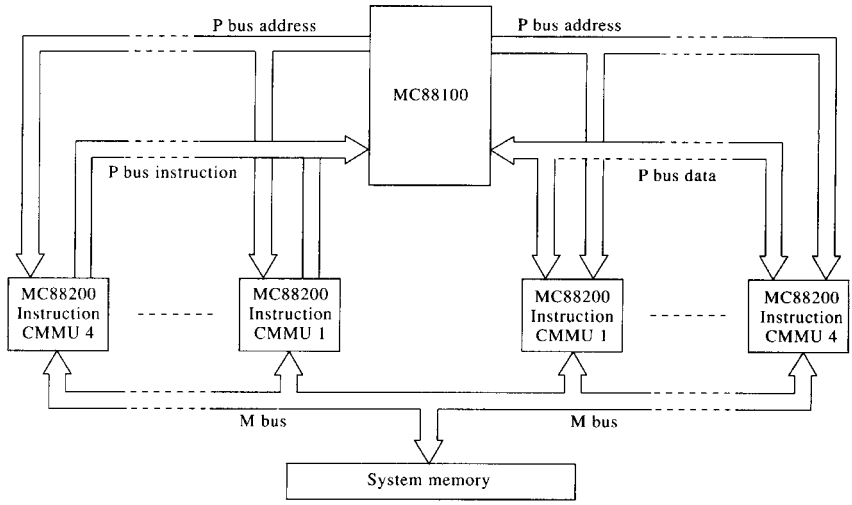



**Figure 9.11** MC88100/MC88200 system diagram example. (Courtesy of Motorola Corporation.)

The cache system capacity is 16 KB. It is a 4-way set-associative cache with 256 sets. There are 4 32-bit words per line of cache. Updating memory is user selectable. The user can either write through or copy back. It is also selectable by area, segment, page, or block. The MC88200 uses the LRU replacement algorithm. Lines can be disabled by line for fault-tolerant operation. Additionally the MC88200 provides a "snoop" capability for coherency.

The MC88200 interfaces the P-bus with the M-bus. The P-bus is a synchronous bus with dedicated address and data lines. It has an 80 MB/s peak throughput at 20 MHz. Checker mode is provided to allow shadowing. The M-bus is a synchronous 32-bit bus. It has multiplexed address and data lines. (Fig. 9.12).

### 9.5.4 Silicon Graphics Inc. MIPS R10000 Architecture

This section is extracted from: *MIPS R10000 User's Manual*, Version 2.0, MIPS Technologies Inc., 1997; *MIPS R10000 Technical Brief*, Version 2.0, MIPS Technologies Inc., 1997; and *ZDNET review of R10000*, December 1997.

The R10000 is a single-chip superscalar RISC microprocessor that is a follow-on to the MIPS RISC processor family that includes, chronolo-

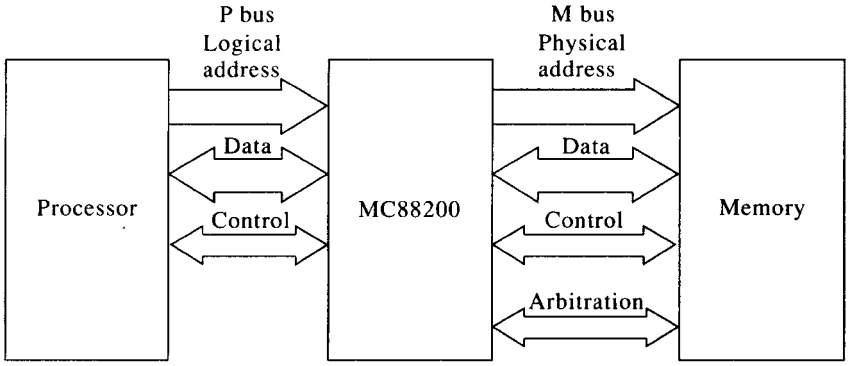



**Figure 9.12** MC88200 logical block diagram. (Courtesy of Motorola Corporation.)

gically, the R2000, R3000, R6000, R4400, and R8000. The integer and floating-point performance of the R10000 makes it ideal for applications such as engineering workstations, scientific computing, 3D graphics workstations, database servers, and multi-user systems. The R10000 uses the MIPS architecture with non-sequential dynamic execution scheduling (ANDES), which supports two integer and two floating-point execute instructions plus one load/store instruction per cycle. The R10000 has the following major features:

- 64-bit MIPS IV instruction set architecture (ISA)
- decoding four instructions each pipeline cycle, appending them to one of three instruction queues
- five execution pipelines connected to separate internal integer and floating-point execution (or functional) units
- dynamic instruction scheduling and out-of-order execution
- speculative instruction issue (also termed "speculative branching")
- precise exception model (exceptions can be traced back to the instruction that caused them)
- non-blocking caches
- separate on-chip 32-Kbyte primary instruction and data caches
- individually optimized secondary cache and system interface ports
- internal controller for the external secondary cache
- internal system interface controller with multiprocessor support.

A block diagram of the processor and its interfaces is shown in Fig. 9.13.

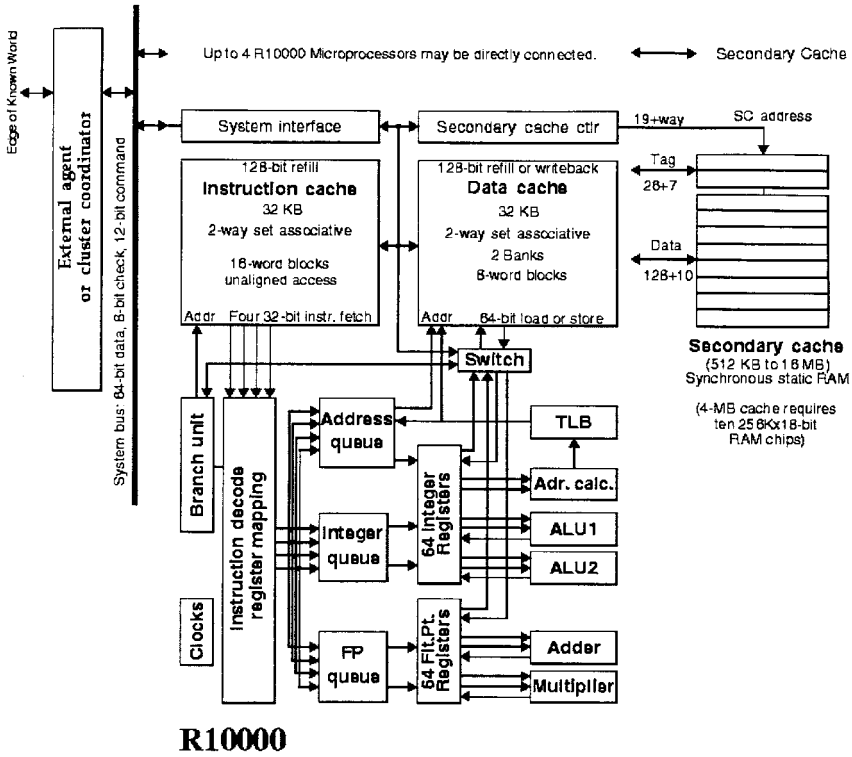



**Figure 9.13** Block diagram of the R10000.

## Instruction Set (MIPS IV)

The R10000 implements the MIPS IV instruction set architecture. MIPS IV is a superset of the MIPS III instruction set architecture and is backward compatible. At a frequency of 200 MHz, the R10000 delivers a peak performance of 800 MIPS with a peak data transfer rate of 3.2 GBytes/second to secondary cache. MIPS has defined an instruction set architecture (ISA), implemented in the following sets of CPU designs:

MIPS I, implemented in the R2000 and R3000
MIPS II, implemented in the R6000
MIPS III, implemented in the R4400
MIPS IV, implemented in the R8000 and R10000 (added prefetch, conditional move I/FP, index load/store FP, etc.).

The original MIPS I CPU ISA has been extended forward three times. Each extension is backward compatible. The ISA extensions are inclusive in the sense that each new architecture level (or version) includes the former levels. The result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

## Superscalar Pipeline

A superscalar processor is one that can fetch, execute and complete more than one instruction in parallel. The R10000 is a four-way super-scalar architecture, which fetches and decodes four instructions per cycle. Each decoded instruction is appended to one of three instruction queues. Each queue can perform dynamic scheduling of instructions. The queues determine the execution order based on the availability of the required execution units. Instructions are initially fetched and decoded in order, but can be executed and completed out-of-order, allowing the processor to have up to 32 instructions in various stages of execution. Instructions are processed in six partially-independent pipelines, as shown in Fig. 9.14. The fetch pipeline reads instructions from the instruction cache, decodes them, renames their registers, and places them in three instruction queues.

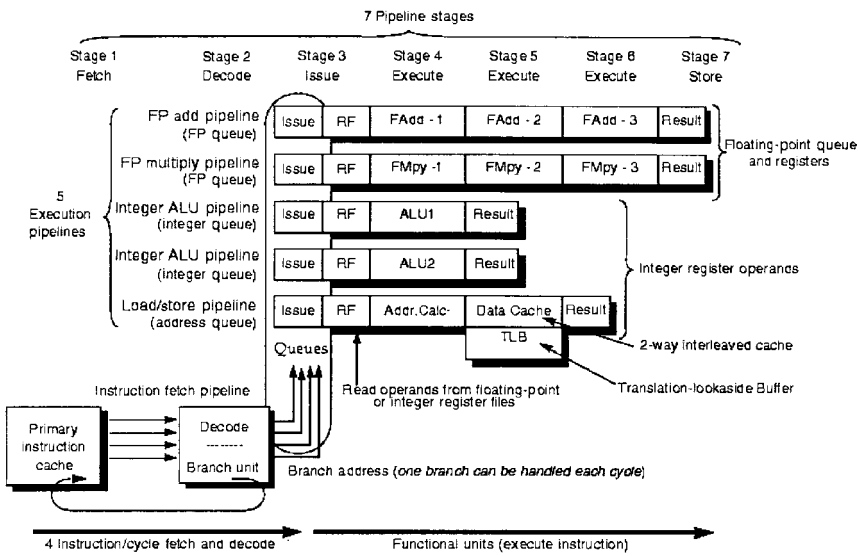



**Figure 9.14** Superscalar pipeline architecture in the R10000.

The instruction queues contain integer, address calculate, and floating-point instructions. From these queues, instructions are dynamically issued to the five pipelined execution units. Each pipeline in R10000 includes stages for fetching (stage 1 in Fig. 9.14), decoding (stage 2), issuing instructions (stage 3), reading register operands (stage 3), executing instructions (stages 4 through 6), and storing results (stage 7).

The processor keeps the decoded instructions in three instruction queues: integer queue, address queue and floating-point queue. These queues allow the processor to fetch instructions at its maximum rate, without stalling because of instruction conflicts or dependencies. Each queue uses instruction tags to keep track of the instruction in each execution pipeline stage. These tags set a done bit in the active list as each instruction is completed.

The integer queue issues instructions to the two integer arithmetic units: ALU1 and ALU2. The integer queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded integer instructions are written into empty entries in no particular order. Instructions remain in this queue only until they have been issued to an ALU. Branch and shift instructions can be issued only to ALU1. Integer multiply and divide instructions can be issued only to ALU2. Other integer instructions can be issued to either ALU. The integer queue controls six dedicated ports to the integer register file: two operand read ports and a destination write port for each ALU.

The floating-point queue issues instructions to the floating-point multiplier and the floating-point adder. The floating-point queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded floating-point instructions are written into empty entries in random order. Instructions remain in this queue only until they have been issued to a floating-point execution unit. The floating-point queue controls six dedicated ports to the floating-point register file: two operand read ports and a destination port for each execution unit. The floating-point queue uses the multiplier's issue port to issue instructions to the square-root and divide units. These instructions also share the multiplier's register ports. The floating-point queue contains simple sequencing logic for multiple-pass instructions such as multiply–add. These instructions require one pass through the multiplier, then one pass through the adder.

The address queue issues instructions to the load/store unit and contains 16 instruction entries. Unlike the other two queues, the address queue is organized as a circular first-in/first-out (FIFO) buffer. A newly decoded load/store instruction is written into the next available sequential empty entry; up to four instructions may be written during each cycle. The FIFO order maintains the program's original instruction sequence so that

memory address dependencies may be easily computed. Instructions remain in this queue until they have graduated; they cannot be deleted immediately after being issued, since the load/store unit may not be able to complete the operation immediately. The address queue contains more complex control logic than the other queues. An issued instruction may fail to complete because of a memory dependency, a cache miss, or a resource conflict; in these cases, the queue must continue to reissue the instruction until it is completed. The address queue has three issue ports:

1. It issues each instruction once to the address calculation unit. This unit uses a 2-stage pipeline to compute the instruction's memory address and to translate it in the TLB. Addresses are stored in the address stack and in the queue's dependency logic. This port controls two dedicated read ports to the integer register file. If the cache is available, it is accessed at the same time as the TLB. A tag check can be performed even if the data array is busy.
2. The address queue can re-issue accesses to the data cache. The queue allocates usage of the four sections of the cache, which consist of the tag and data sections of the two cache banks. Load and store instructions begin with a tag check cycle, which checks to see if the desired address is already in cache. If it is not, a refill operation is initiated, and this instruction waits until it has completed. Load instructions also read and align a doubleword value from the data array. This access may be either concurrent to or subsequent to the tag check. If the data is present and no dependencies exist, the instruction is marked done in the queue.
3. The address queue can issue store instructions to the data cache. A store instruction may not modify the data cache until it graduates. Only one store can graduate per cycle, but it may be anywhere within the four oldest instructions, if all previous instructions are already completed.

The access and store ports share four register file ports (integer read and write, floating-point read and write). These shared ports are also used for jump and link and jump register instructions, and for move instructions between the integer and register files.

The three instruction queues can issue one new instruction per cycle to each of the five execution pipelines:

- integer queue issues instructions to the two integer ALU pipelines
- address queue issues one instruction to the load/store unit pipeline
- floating-point queue issues instructions to the floating-point adder and multiplier pipelines.

A sixth pipeline, the fetch pipeline, reads and decodes instructions from the instruction cache.

The 64-bit integer pipeline has the following characteristics:

- 16-entry integer instruction queue that dynamically issues instructions
- 64-bit 64-location integer physical register file, with seven read and three write ports
- 64-bit arithmetic logic units:
  - arithmetic-logic unit, shifter, and integer branch comparator
  - arithmetic-logic unit, integer multiplier, and divider.

The load/store pipeline has the following characteristics:

- 16-entry address queue that dynamically issues instructions, and uses the integer register file for base and index registers
- 16-entry address stack for use by non-blocking loads and stores
- 44-bit virtual address calculation unit
- 64-entry fully associative translation-lookaside buffer (TLB), which converts virtual addresses to physical addresses, using a 40-bit physical address. Each entry maps two pages, with sizes ranging from 4 KB to 16 MB, in powers of 4.

The 64-bit floating-point pipeline has the following characteristics:

- 16-entry instruction queue, with dynamic issue
- 64-bit 64-location floating-point physical register file, with five read and three write ports (32 logical registers)
- 64-bit parallel multiply unit (3-cycle pipeline with 2-cycle latency) which also performs move instructions
- 64-bit add unit (3-cycle pipeline with 2-cycle latency) which handles addition, subtraction and miscellaneous floating-point operations
- separate 64-bit divide and square-root units which can operate concurrently (these units share their issue and completion logic with the floating-point multiplier).

### Functional Units

The five execution pipelines allow overlapped instruction execution by issuing instructions to the following five functional units: two integer ALUs (ALU1 and ALU2), load/store unit (address calculate), floating-point adder and floating-point multiplier. There are also three ''iterative'' units to compute more complex results:

The integer multiply and divide operations are performed by an integer multiply/divide execution unit; these instructions are issued to ALU2. ALU2 remains busy for the duration of the divide. Floating-point divides are performed by the divide execution unit; these instructions are issued to the floating-point multiplier. Floating-point square root are performed by the square-root execution unit; these instructions are issued to the floating-point multiplier.

The instruction decode and rename unit has the following characteristics:

- processes four instructions in parallel
- replaces logical register numbers with physical register numbers (register renaming)
- maps integer registers into a 33-word-by-6-bit mapping table that has 4 write and 12 read ports
- maps floating-point registers into a 32-word-by-6-bit mapping table that has 4 write and 16 read ports
- has a 32-entry active list of all instructions within the pipeline.

The branch unit has the following characteristics:

- Allows one branch per cycle.
- Conditional branches can be executed speculatively, up to 4-deep.
- 44-bit adder to compute branch addresses.
- The branch return cache contains four instructions following a subroutine call, for rapid use when returning from leaf subroutines.
- The program trace RAM stores the program counter for each instruction in the pipeline.

## Pipeline Stages

In Stage 1, the procesor fetches four instructions each cycle, independent of their alignment in the instruction cache – except that the processor cannot fetch across a 16-word cache block boundary. These words are then aligned in the four-word instruction register. If any instructions were left from the previous decode cycle, they are merged with new words from the instruction cache to fill the instruction register.

In Stage 2, the four instructions in the instruction register are decoded and renamed. (Renaming determines any dependencies between instructions and provides precise exception handling.) When renamed, the logical registers referenced in an instruction are mapped to physical registers. Integer and floating-point registers are renamed independently. A logical register is

mapped to a new physical register whenever that logical register is the destination of an instruction.

Thus, when an instruction places a new value in a logical register, that logical register is renamed (mapped) to a new physical register, while its previous value is retained in the old physical register.

As each instruction is renamed, its logical register numbers are compared to determine if any dependencies exist between the four instructions decoded during this cycle. After the physical register numbers become known, the physical register busy table indicates whether or not each operand is valid. The renamed instructions are loaded into integer or floating-point instruction queues.

Only one branch instruction can be executed during Stage 2. If the instruction register contains a second branch instruction, this branch is not decoded until the next cycle. The branch unit determines the next address for the program counter; if a branch is taken and then reversed, the branch resume cache provides the instructions to be decoded during the next cycle.

In Stage 3, decoded instruction are written into the queues. Stage 3 is also the start of each of the five execution pipelines.

In Stages 4–6, instructions are executed in the various functional units. These units and their execution process are described below.

*Floating-point multiplier (three-stage pipeline)*: Single- or double-precision multiply and conditional move operations are executed in this unit with a one-cycle latency and a one-cycle repeat rate. The multiplication is completed during the first two cycles; the third cycle is used to pack and transfer the result.

*Floating-point divide and square-root units*: Single- or double-precision division and square-root operations can be executed in parallel by separate units. These units share their issue and completion logic with the floating-point multiplier.

*Floating-point adder (three-stage pipeline)*: Single- or double-precision add, subtract, compare, or convert operations are executed with a two-cycle latency and a one-cycle repeat rate. Although a final result is not calculated until the third pipeline stage, internal bypass paths set a two-cycle latency for dependent add or multiply instructions.

*Integer ALU1 (one-stage pipeline)*: Integer add, subtract, shift, and logic operations are executed with a one-cycle latency and a one-cycle repeat rate. This ALU also verifies predictions made for branches that are conditional on integer register values.

*Integer ALU2 (one-stage pipeline)*: Integer add, subtract, and logic operations are executed with a one-cycle latency and a one-cycle repeat rate. Integer multiply and divide operations take more than one cycle.

A single memory address can be calculated every cycle for use by either an integer or a floating-point load or store instruction. Address calculation and load operations can be calculated out of program order. The calculated address is translated from a 44-bit virtual address into a 40-bit physical address using a translation-lookaside buffer. The TLB contains 64 entries, each of which can translate two pages. Each entry can select a page size ranging from 4 KB to 16 MB, inclusive, in powers of 4, as shown in Fig. 9.15.

Load instructions have a two-cycle latency if the addressed data is already within the data cache. Store instructions do not modify the data cache or memory until they graduate.

## Cache

The R10000 contains an on-chip cache consisting of 32 KB of primary data cache and a separate 32 KB instruction cache. This is considered a large on-chip cache. This processor controls an off-chip secondary cache that can range in size from 512 KB to 16 MB.

The primary data cache consists of two equal 16 KB banks. The data cache is two-way interleaved and each of the two banks is two-way set associative using a least recently used replacement algorithm. The data cache line size is 32 bytes and the data cache contains a fixed block size of 8 words.

The interleaved data cache design was used because access times of most currently available random access memory is long relative to processor cycle times. The interleaved data cache allows the memory requests to be overlapped, which in turn allows the ability to hide more of the access and recovery times of each bank. To speed access delays the processor can:

- Execute up to 16 load and store instructions speculatively and out of order, using non-blocking primary and secondary caches. Meaning, it looks ahead in its instruction stream to find load and store instructions which can be executed early; if the addressed data blocks are not in the primary cache, the processor initiates cache refills as soon as possible.

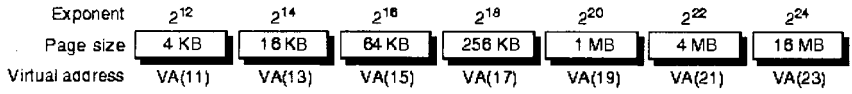



**Figure 9.15** TLB page sizes.

- If a speculatively executed load initiates a cache refill, the refill is completed even if the load instruction is aborted.
- The external interface gives priority to its refill and interrogate operations. When the primary cache is refilled, any required data can be streamed directly to waiting load instructions.
- The external interface can handle up to four non-blocking memory accesses to secondary cache and main memory.

Because of an interleaved data cache increased complexity is required to support them. This complexity begins with each cache bank containing independent tag and data arrays. These four sections can be allocated separately to achieve high utilization. Five separate circuits compete for cache bandwidth. These circuits are the address calculate, tag check, load unit, store unit and external interface.

The data cache is non-blocking. This allows the cache accesses to continue even though a cache miss has occurred. This is important to the cache performance because it gives the data cache the ability to stack memory references by queuing up multiple cache misses and servicing them simultaneously.

The data cache uses a write back protocol, which means a cache store writes data into the cache instead of writing it directly to memory. When data is written to the data cache, it is tagged as a dirty block. Prior to this dirty block being replaced by a new frame it is written back to the off-chip secondary cache. The secondary cache writes back to the main memory. This protocol is used to maintain data consistency. Note that the data cache is written back prior to the secondary cache writing back to the main memory.

Because the data cache is a subset of the secondary cache the data cache is said to be inconsistent when it has been modified from the corresponding data in the secondary cache. The data cache can be in only one of the following four states at any given time: *invalid, clean exclusive, dirty exclusive,* and *shared.*

The processor requires the cache blocks to have a single owner at all times. Thus the processor adheres to certain ownership rules:

- The processor assumes ownership of a cache block if the state of the block becomes dirty exclusive.
- For a processor upgrade request, the processor assumes ownership of the block after receiving an external ACK completion response.
- The processor gives up ownership of a cache block if the state of the cache block changes to invalid, clean exclusive, or shared.
- Clean exclusive and shared cache blocks are always considered to be owned by memory.

The events that trigger a change in the state of a data cache block included in the following events:

1. Primary data cache read/write miss
2. Primary data cache hit
3. Subset enforcement
4. A cache instruction
5. External intervention shared request
6. Intervention exclusive request.

The secondary cache is located off-chip and can range in size from 512 KB to 16 MB. It is interfaced with the R10000 by a 128-bit data bus, which can operate at a maximum of 200 MHz, yielding a maximum transfer rate of 3.2 GB per second. This dedicated cache bus cannot be interrupted by any other bus traffic in the system. Thus when a cache miss occurs the access to the secondary cache is immediate. Therefore it is true to say that the secondary cache interface approaches zero wait state peformance. This means that when the cache receives a data request from the processor it will always be able to return data in the following clock cycle. There is no external interface circuitry required for the secondary cache system. The secondary cache maintains many of the same features as the primary data cache. It is two-way set associative and uses a least recently used replacement algorithm. It also uses the write back protocol and can be in one of the following four states: *invalid, clean exclusive, dirty exclusive*, and *shared*.

The events that trigger the changing of a secondary cache block are the same as the primary data cache except for events (2)–(4) below:

1. Primary data cache read/write miss
2. Data cache write hit to a *shared* or *clean exclusive* block
3. Secondary cache read miss
4. Secondary cache write hit to a *shared* or *clean exclusive* block
5. A cache instruction
6. External intervention shared request
7. Intervention exclusive request
8. Invalidate request.

The R10000 has a 32 KB on-chip instruction cache that is two-way set associative. It has a fixed block size of 16 words, a line size of 64 bytes and uses the least recently used replacement algorithm. At any given time the instruction cache may be in one of two states, valid or invalid. The following are the events that cause the instruction cache block to change states:

1. A primary instruction cache read miss
2. Subset property enforcement

3. Any of various cache instructions
4. External intervention exclusive and invalidate requests.

The behavior of the processor when executing load and store instructions is determined by the cache algorithm specified for the accessed address. The processor supports five different cache algorithms:

- Uncached
- Cacheable non-coherent
- Cacheable coherent exclusive
- Cacheable coherent exclusive on write
- Uncached accelerated.

The loads and stores under the uncached cache algorithm bypass the primary and secondary caches. Under the cacheable non-coherent cache algorithm, load and store secondary cache misses result in processor non-coherent block read requests. Under the cacheable coherent exclusive cache algorithm, load and store secondary cache misses result in processor coherent block read exclusive requests. Such processor requests indicate to external agents containing caches that a coherency check must be performed and that the cache block must be returned in an exclusive state. The cacheable coherent exclusive on write cache algorithm is similar to the cacheable coherent exclusive cache algorithm except that load secondary cache misses result in processor coherent block read shared requests. The R10000 implements a new cache algorithm, uncached accelerated. This allows the kernel to mark the TLB entries for certain regions of the physical address space, or certain blocks of data, as uncached. After signaling the hardware it may now gather a number of uncached writes together as a series of writes to the same address or sequential writes to all addresses in the block. These are put to an uncached accelerated buffer and then issued to the system interface as processor block write requests. The uncached accelerated algorithm differs from the uncached algorithm in that block write gathering is not performed.

### Processor Operating Modes

The three processor operating modes are listed in order of decreasing system privilege:

*Kernel mode* (highest system privilege): can access and change any register. The innermost core of the operating system runs in kernel mode.

*Supervisor mode*: has fewer privileges and is used for less critical sections of the operating system.

*User mode* (lowest system privilege): prevents users from interfering with one another.

Selection between the three modes can be made by the operating system (when in kernel mode) by writing into the status register's KSU field. The processor is forced into kernel mode when the processor is handling an error (the ERL bit is set) or an exception (EXL bit is set). Figure 9.16 shows the selection of operating modes with respect to the KSU, EXU, and ERL bits. It also shows how different instruction sets and addressing modes are enabled by the status register's XX, UX, SX, and KX bits. In kernel mode the KX bit allows 64-bit addressing; all instructions are always valid. In supervisor mode, the SX bit allows 64-bit addressing and the MIPS III instructions. MIPS IV ISA is enabled all the time in supervisor mode. In user mode the UX bit allows 64-bit addressing and the MIPS III instructions; the XX bit allows the new MIPS IV instruction.

The processor uses either 32-bit or 64-bit address spaces, depending on the operating and addressing modes set by the status register. The processor uses the following addresses: virtual address VA and region bits VA. If a region is mapped, virtual addresses are translated in the TLB. Bits VS are not translated in the TLB and are sign extensions of bit VA.

In both 32-bit and 64-bit address mode, the memory address space is divided into many regions as shown in Fig. 9.17. Each region has specific characteristics and uses. The user can access only the useg region in 32-bit mode, xuseg in 64-bit mode as shown in the table above. The supervisor can

| XX 31 | KX 7 | SX 6 | UX 5 | KSU 4:3 | ERL 2 | EXL 1 | Description | ISA* III | ISA† IV | Addressing mode 32-bit/64-bit |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | - | 0 | 10 | 0 | 0 | User mode | No | No | 32 |
| 1 | - | - | 0 | 10 | 0 | 0 | | No | Yes | 32 |
| 0 | - | - | 1 | 10 | 0 | 0 | | Yes | No | 64 |
| 1 | - | - | 1 | 10 | 0 | 0 | | Yes | Yes | 64 |
| - | - | 0 | - | 01 | 0 | 0 | Supervisor mode | No | Yes | 32 |
| - | - | 1 | - | 01 | 0 | 0 | | Yes | Yes | 64 |
| - | 0 | - | - | 00 | 0 | 0 | Kernel mode | Yes | Yes | 32 |
| - | 1 | - | - | 00 | 0 | 0 | | Yes | Yes | 64 |
| - | 0 | - | - | - | 0 | 1 | Exception level | Yes | Yes | 32 |
| - | 1 | - | - | - | 0 | 1 | | Yes | Yes | 64 |
| - | 0 | - | - | - | 1 | X | Error level | Yes | Yes | 32 |
| - | 1 | - | - | - | 1 | X | | Yes | Yes | 64 |

*No means the ISA is disabled; Yes means the ISA is enabled.

†Dashes (-) are "don't care."

**Figure 9.16** Operating mode versus addressing modes.

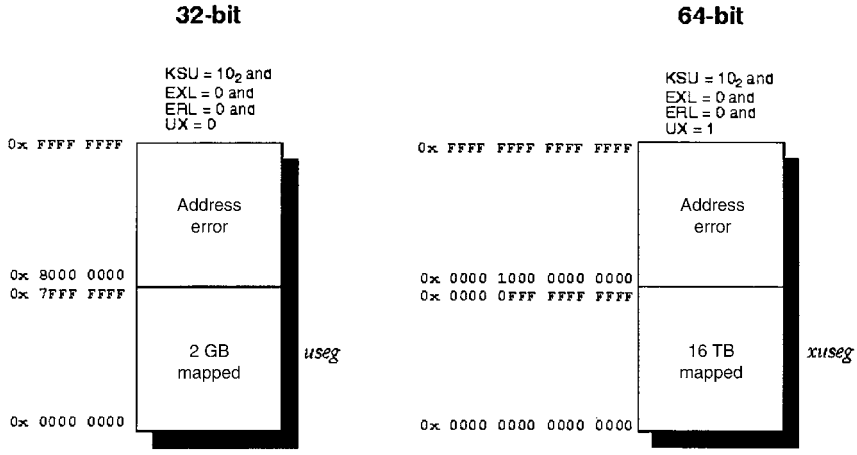



**Figure 9.17** User mode virtual address space.

access user region as well as sseg (in 32-bit mode) or xsseg and csseg (in 64-bit mode) shown in Fig. 9.17. The kernel can access all regions except those restricted because bits VA are not implemented in the TLB.

In user mode, a single uniform virtual address space-labeled user segment is available; its size is: 2 GB in 32-bit mode (useg), 16 TB in 64-bit mode (xuseg). When UX = 0 in the status register, user mode addressing is compatible with the 32-bit addressing model and a 2 GB user address space is available labeled useg. All valid user mode virtual addresses have their most significant bit cleared to 0. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception. When UX = 1 in the status register, user mode addressing is extended to the 64-bit model shown in Fig. 9.17. In 64-bit user mode, the processor provides a single, uniform virtual address space of $2^{44}$ bytes labeled xuseg. All valid user mode virtual addresses have bits 63:44 equal to 0; an attempt to reference an address with bits 63:44 not equal to 0 causes an address error exception.

Supervisor mode is designed for layered operating systems in which a true kernel runs in processor kernel mode, and the rest of the operating system runs in supervisor mode. In this mode, when SX = 0 in the status register and the most significant bit of the 32-bit virtual address is set to 0, the suseg virtual address space is selected; it covers the full 231 bytes of the current user address space. The virtual address is extended with the contents of the 8-bit field to form a unique virtual address. When SX = 0 in the status register and the three most significant bits of the 32-bit virtual address are

$110_2$, the sseg virtual address space is selected; it covers $2^{29}$ bytes of the current supervisor address space (Fig. 9.18).

The processor operates in kernel mode when the status register contains the kernel mode bit values as shown in Fig. 9.19. Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address. In this mode, when KX = 0, in the status register and the most significant bit of the virtual address, A31, is cleared, the 32-bit kuseg virtual address space is selected. It covers the full 2 GB of the current user address space. When KX = 0 in the status register and the most significant three bits of the virtual address are $100_2$, 32-bit kseg() virtual address space is selected. References to kseg() are not mapped through the TLB. When KX = 0 in the status register and the most significant three bits of the 32-bit virtual address are $10_2$, the ksseg virtual address space is selected; it is the current 512 MB supervisor virtual space. When KX = 1 in the status register and bits 63:62 of the 64-bit virtual address are $10_2$, the xkphys virtual address space is selected; it is a set of eight kernel physical spaces. Each kernel physical space contains either one or four $2^{40}$-byte physical pages. References to this space are not mapped; the physical address selected is taken directly from bits 39:0 of the virtual address. Bits 61:59 of the virtual address specify the cache

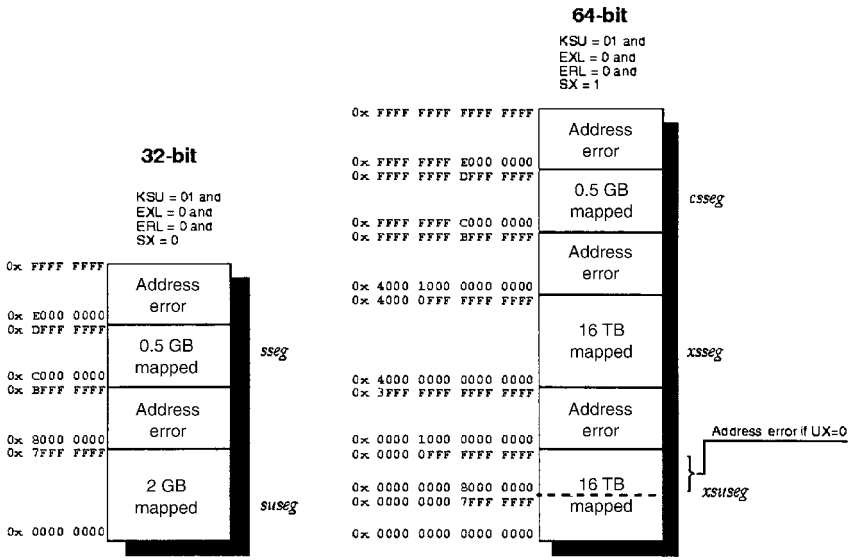



**Figure 9.18** Supervisor mode address space

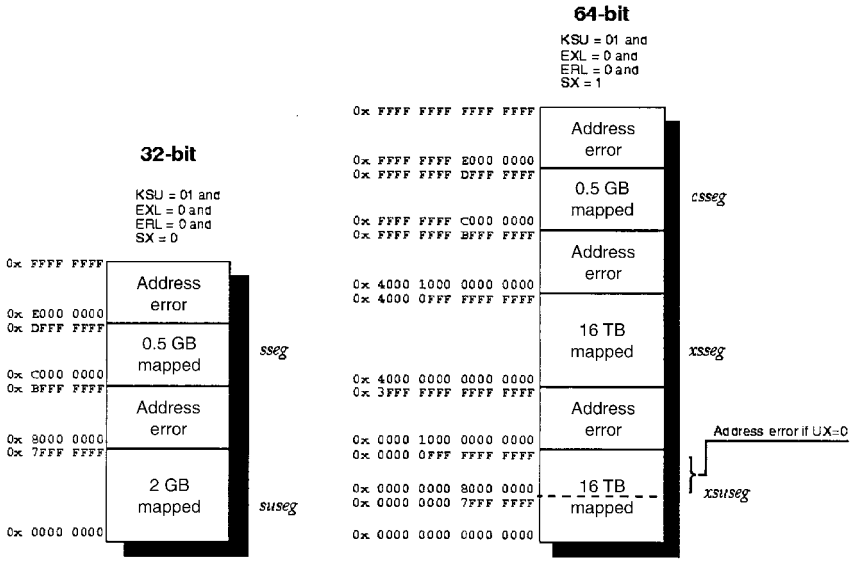



**Figure 9.19** Kernel mode physical space.

algorithm. If the cache algorithm is either uncached or uncached accelerated the space contains four physical pages; access to addresses whose bits 56:40 are not equal to 0 cause an address error exception.

Virtual to physical address translations are maintained by the operating system, using page tables in memory. A subset of these translations are loaded into a hardware buffer (TLB). The TLB contains 64 entries, each of which maps a pair of virtual pages. Formats of TLB entries are shown in Fig. 9.20. The cache Algorithm fields of the TLB, entrylo0, entrylo1 and Config registers indicate how data is cached.

Figure 9.20 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the entryhi, Entrylo0, entrylo1 or PageMask registers.

Because a 64-bit is unnecessarily large, only the low 44 address bits are translated. The high two virtual address bits (bits 63:62) select between user, supervisor and kernel address space. The intermediate address bits (61:44) must either be all zeros or all ones depending on the address region. For data cache accesses, the joint TLB translates addresses from the address calculate unit. For instruction accesses, the JTLB translates the PC address if it misses in the instruction TLB. That entry is copied into the ITLB for subsequent accesses.

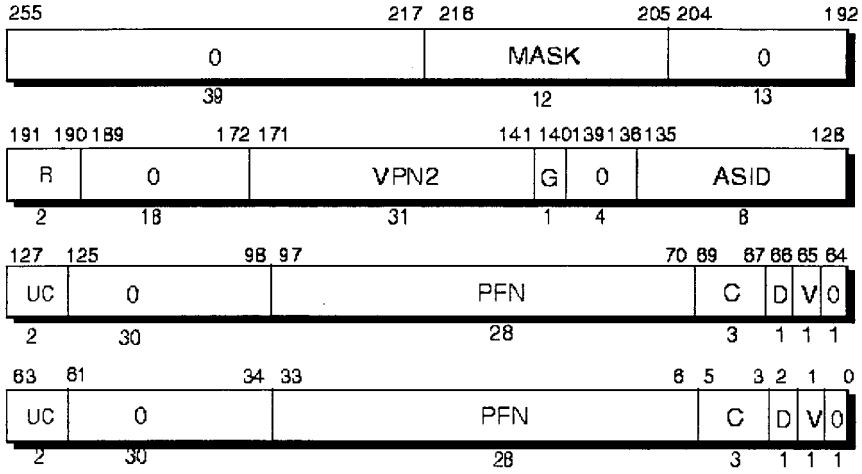



**Figure 9.20** TLB entries.

Each independent task or process has a separate address space, assigned a unique 8-bit address space identifier (ASID). This identifier is stored with each TLB entry to distinguish between entries loaded for different processes. The ASID allows the processor to move from one process to another (called a context switch) without having to invalidate TLB entries.

## Floating-Point Units

The R10000 contains two primary floating-point units. The adder unit handles add operations and the multiply unit handles multiply operations. In addition, two secondary floating-point units exist which handle long-latency operations such as divide and square root.

Addition, subtraction, and conversion instructions have a two-cycle latency and a one-cycle repeat rate and are handled within the adder unit. Instructions which convert integer values to single-precision floating-point values have a four-cycle latency as they must pass through the adder twice. The adder is busy during the second cycle after the instruction is issued.

All floating-point multiply operations execute with a two-cycle latency and a one-cycle repeat rate and are handled within the multiplier unit. The multiplier performs multiply operations. The floating-point divide and square root units perform calculations using iterative algorithms. These units are not pipelined and cannot begin another operation until the current operation is completed. Thus, the repeat rate approximately equals the

latency. The ports of the multiplier are shared with the divide and square root units. A cycle is lost at the beginning of the operation (to fetch the operand) and at the end (to store the result).

The floating-point multiply–add operation, which occurs frequently, is computed using separate multiply and add operations. The multiply–add instruction (MADD) has a four-cycle latency and a one-cycle repeat rate. The combined instruction improves performance by eliminating the fetching and decoding of an extra instruction. The divide and square root units use separate circuitry and can be operated simultaneously. However, the floating-point queue cannot issue both instructions during the same cycle.

The floating-point add, multiply, divide, and square-root units read their operands and store their results in the floating-point register file. Values are loaded to or stored from the register file by the load/store and move units.

A logic diagram of floating-point operations is shown in Fig. 9.21 in which data and instructions are read from the secondary cache into the primary caches, and then into the processor. There they are decoded and appended to the floating-point queue, and passed into the FP register file where each is dynamically issued to the appropriate functional unit. After execution in the functional unit, results are stored, through the register file, in the primary data cache.

The floating-point queue can issue one instruction to the adder unit and one instruction to the multiplier unit. The adder and multiplier each has two dedicated read ports and a dedicated write port in the floating-point register file. Because of their low repeat rates, the divide and square-root units do not have their own issue port. Instead, they decode instructions issued to the multiplier unit, using its operand registers and bypass logic. They appropriate a second cycle later for storing their result.

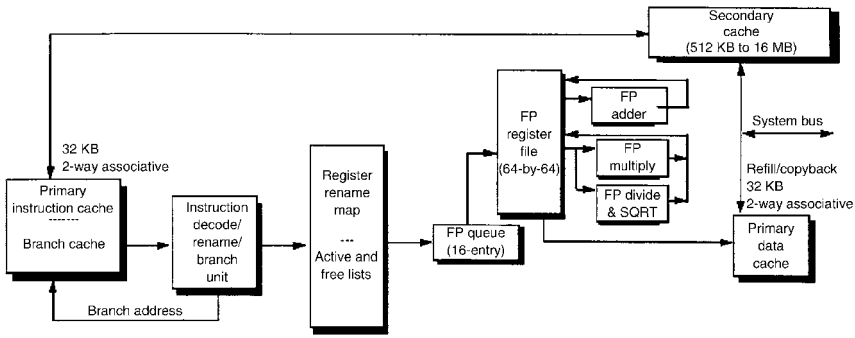



**Figure 9.21** Logical diagram of FP operations.

When an instruction is issued, up to two operands are read from dedicated read ports in the floating-point register file. After the operation has been completed, the result can be written back into the register file using a dedicated write port. For the add and multiply units, this write occurs four cycles after its operands were read.

The control of floating-point execution is shared by the following units:

- The floating-point queue determines operand dependencies and dynamically issues instructions to the execution units. It also controls the destination registers and register bypass.
- The execution units control the arithmetic operations and generate status.
- The graduate unit saves the status until the instructions graduate, and then it updates the *floating-point status* register.

The floating-point unit is the hardware implementation of Coprocessor 1 in the MIPS IV instruction set architecture. The MIPS IV ISA defines 32 logical floating-point general registers (FGRs), as shown in Fig. 9.22. Each FGR is 64 bits wide and can hold either 32-bit single-precision or 64-bit double-precision values. The hardware actually contains 64 physical 64-bit registers in the floating-point register file, from which the 32 logical registers are taken.

Floating-point instructions use a 5-bit logical number to select an individual FGR. These logical numbers are mapped to physical registers by the rename unit (in pipeline stage 2), before the floating-point unit executes them. Physical registers are selected using 6-bit addresses.

The *FR* bit (26) in the status register determines the number of logical floating-point registers. When FR = 1, floating-point load and stores operate as follows:

- Single-precision operands are read from the low half of a register, leaving the upper half ignored. Single-precision results are written into the low half of the register. The high half of the result register is architecturally undefined; in the R10000 implementation, it is set to zero.
- Double-precision arithmetic operations use the entire 64-bit contents of each operand or result register.

Because of register renaming, every new result is written into a temporary register, and conditional move instructions select between a new operand and the previous old value. The high half of the destination register of a single-precision conditional move instruction is undefined, even if no move occurs.

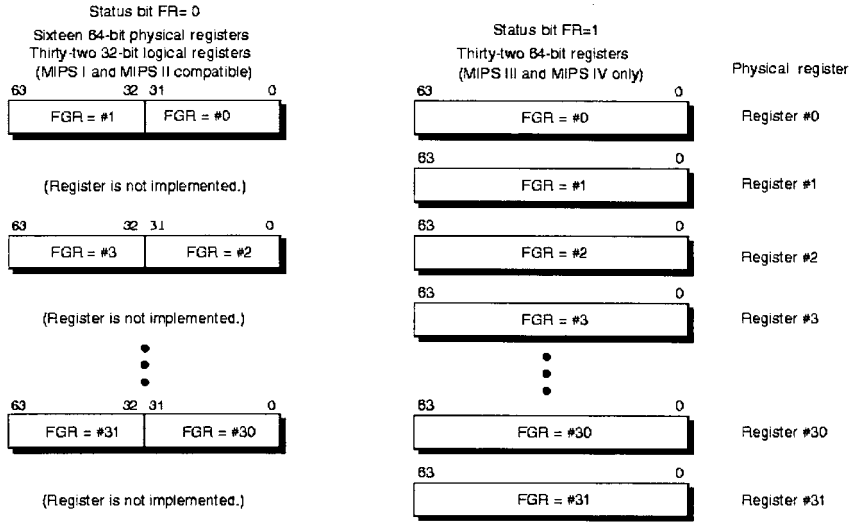



**Figure 9.22** Floating-point registers in 16-bit mode.

The load/store unit consists of the address queue, address calculation unit, translation lookaside buffer (TLB), address stack, store buffer, and primary data cache. The load/store unit performs load, store, prefetch, and cache instructions. All load or store instructions begin with a three-cycle sequence which issues the instruction, calculates its virtual address, and translates the virtual address to physical. The address is translated only once during the operation. The data cache is accessed and the required data transfer is completed provided there was a primary data cache hit. If there is a cache miss, or if the necessary shared register ports are busy, the data cache and data cache tag access must be repeated after the data is obtained from either the secondary cache or main memory.

The TLB contains 64 entries and translates virtual addresses to physical addresses. The virtual address can originate from either the address calculation unit or the program counter (PC).

## Interrupts

Figure 9.23 shows the interrupt structure of the R10000.

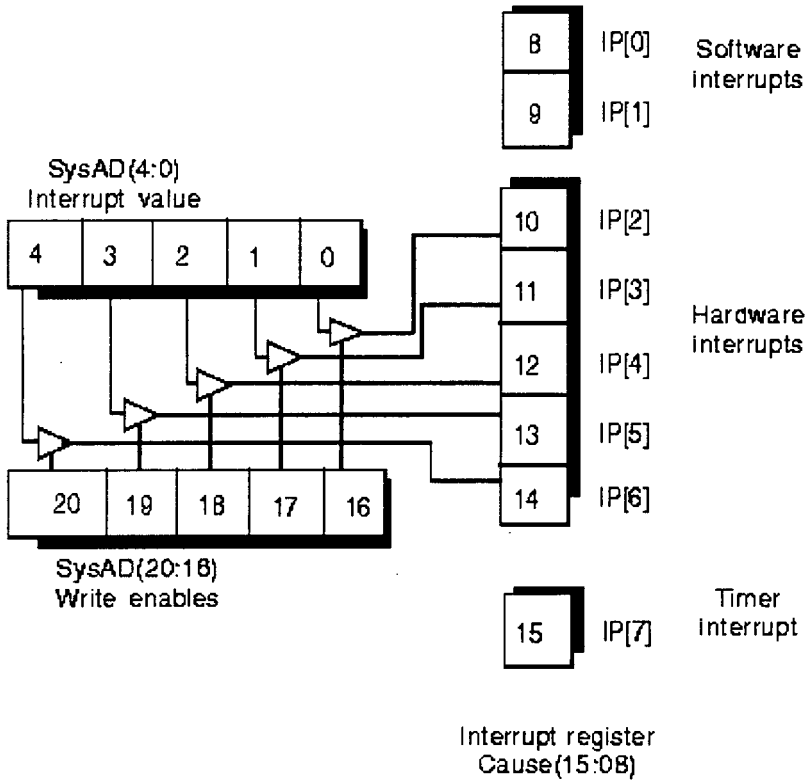



**Figure 9.23** Interrupts in R10000.

## 9.6 SUMMARY

The major parameters of concern in the design of control units are speed, cost, complexity, and flexibility. Hardwired control units offer higher speeds than microprogrammed control units, while the MCUs offer better flexibility. Speed enhancement techniques applicable to both types of control units were discussed in this chapter. Commonly used microinstruction formats were introduced along with their speed-cost tradeoffs. Examples of contemporary architectures were also provided.

## REFERENCES

*Computer*, New York, NY: IEEE Computer Society. Published monthly.
*Computer Design*, Northbrook, IL: PennWell Publishing. Published monthly.

Cragon, G. C., *Memory Systems and Pipelined Processors*, Boston, MA: Jones and Bartlett, 1996.
Culler, D. E., Singh, J. and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA: Morgan Kaufmann, 1998.
*Electronic Design*, Hasbrouck Heights, NJ: Penton Publishing. Published twice monthly.
Hill, M. D., Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.
Husson, S. S. Microprogramming Principles and Practice. Englewood Cliffs, N.J.: Prentice-Hall, 1970.
INTEL 8080 Microcomputer Systems Users Manual. Santa Clara, CA: INTEL Corporation, 1977.
*MIPS R10000 User's Manual*, Version 2.0, MIPS Technologies Inc., 1997.
*MIPS R10000 Technical Brief*, Version 2.0, MIPS Technologies Inc., 1997.
Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.
Stone, H. S. *High-Performance Computer Architectures*. Reading, MA: Addison Wesley, 1987.
Vax Hardware Handbook, Maynard, MA: Digital Equipment Corporation, 1979.
Wilson, R. "Motorola Unveils New RISC Microprocessor Flagship." *Computer Design*, May 1988, pp. 21–32.
ZDNET Review of MIPS R10000, December 1997.

## PROBLEMS

9.1 Study the following architectural features of a micro-, a mini-, and a large-scale processor with an HCU you have access to:
   a. Processor cycle time
   b. Number of clock phases and clock frequency
   c. Minor and major cycle details
   d. Speedup techniques used.

9.2 Study the following architectural features of a micro-, a mini-, and a large-scale processor with an MCU you have access to:
   a. Processor cycle time
   b. CROM size, access time, and organization
   c. Microinstruction format
   d. Speedup techniques used.

9.3 It is required that the LDA instruction cycle use a maximum of two major cycles. Derive the microinstruction sequence and list any changes needed to the bus structure.

9.4 Write the microprogram for the LDA instruction in Problem 9.3. List any changes needed for the microinstruction format.

9.5 Assume that an HCU needs to be designed for LDA, STA, TCA, and ADD instructions only of ASC. Examine the microoperation sequences for these instructions to determine the optimum length for the major cycle if (a) only equal length major cycles are allowed (b) major cycles can be of varying length.

9.6 In the microprogram for ASC (Table 5.6), the sequence for indirect address computation is repeated for each ASC instruction requiring that computation. Suppose that the indirect address computation is made into a subroutine called by ASC instructions as required, rewrite the microprogram for LDA. What is the effect on the execution time of LDA if indirect address is (a) called for? (b) not called for?

9.7 List the changes needed to convert ASC HCU to include three modules – fetch, defer, execute – working in a pipeline mode. Assume that only LDA, STA, SHR, TCA, LDX, and HLT instructions are needed.

9.8 Repeat Problem 9.7 listing changes needed for ASC MCU.

9.9 ASC MCU currently employs two types of microinstructions. Determine the microinstruction format needed if only a single type of microinstruction is allowed.

9.10 Repeat Problem 9.9 assuming that only horizontal microinstruction is allowed.

9.11 A program consists of $N$ instructions. Assume that all the $N$ instruction cycles require the fetch, decode and execute phases. If each phase consumes one time unit, compute the serial execution time of the program. What is the speedup, if a pipelined control unit with three stages is used?

9.12 Assume that ASC uses a three-stage pipelined control unit. show the (time line) trace of the following program.

```
LDA   X
ADD   Y
STA   Z
HLT
```

9.13 Repeat Problem 9.12 assuming a control unit with the following stages: fetch, decode, compute address, fetch operands, execute, and store results.

9.14 Compare the instruction sets of Intel 8080 and Intel Pentium II with respect to instruction execution speeds and modes of execution.

9.15 Compare the instruction sets of Intel Pentium II and MIPS R10000 with respect to instruction set, instruction execution speeds and modes of execution.

# 10
# Arithmetic/Logic Unit Enhancement

The ASC arithmetic/logic unit (ALU) was designed to perform addition, complementation, and single-bit shift operations on 16-bit binary data, in parallel. The four basic arithmetic operations (addition, subtraction, multiplication, and division) can be performed using this ALU and appropriate software routines as described in Chapter 4. Advances in IC technology have made it practical to implement most of the ALU functions required in hardware, thereby minimizing the software implementation and increasing the speed. In this chapter, we will describe a number of enhancements that make the ASC ALU resemble a practical ALU.

The majority of modern-day processors use *bit-parallel* ALUs. It is possible to use a *bit-serial* ALU where slow operations are acceptable. For example, in a calculator IC, serial arithmetic may be adequate due to the slow nature of human interaction with the calculator. In building processors on single ICs, a considerable amount of silicon "real estate" can be conserved by using a serial ALU. The silicon area thus saved can be used in implementing other complex operations that may be required.

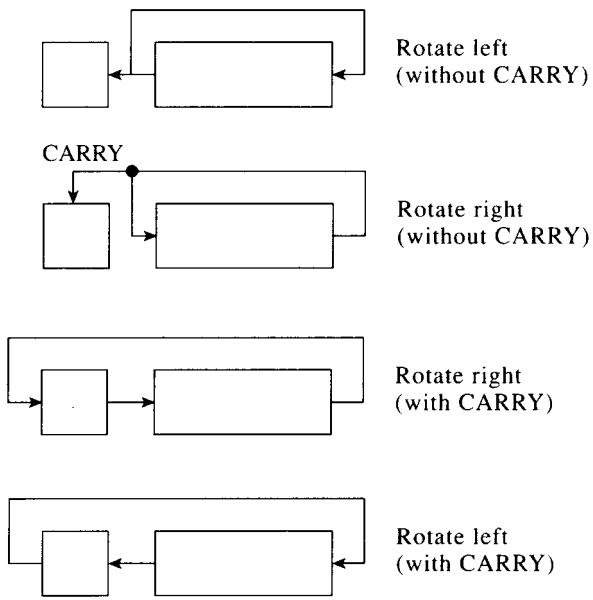
As discussed in Chapter 7, the most common operands an ALU deals with are: binary (fixed- and floating-point), decimal, and character strings. We will describe the ALU enhancements that facilitate fixed-point binary and logical data manipulation in the next section. Section 10.2 deals with decimal arithmetic. Section 10.3 describes the pipelining concept as applied to ALUs. It is now common to see either multiple functional units within an ALU or multiple ALUs within the CPU. Section 10.4 outlines the use of multiple units to enable the parallelism of operations, thereby achieving higher speeds. Section 10.5 provides some details of several commercially available systems.

## 10.1 LOGICAL AND FIXED-POINT BINARY OPERATIONS

ASC ALU is designed to perform a single-bit right or left shift. Other types of shift operations are useful in practice. The parallel binary adder used in ASC ALU is slow, since the carry has to ripple from the least significant bit to the most significant bit before the addition is complete. Several algorithms for fast addition have been devised. We will describe one such algorithm in this section. Various multiplication and division schemes have been devised; we will describe the popular ones.

### 10.1.1 Logical Operations

The single-bit shifts performed by the ASC ALU are called *arithmetic shift* operations since the results of shift conformed to the 2s complement arithmetic used by the ALU. In practical ALUs, it is common to see the following additional types of arithmetic shift operations:




Logical operations such as AND, OR, EXCLUSIVE OR, and NOT are also useful ALU functions. The operands for these operations could be the contents of registers and/or memory locations. For instance, in ASC,

AND Z

might imply bit-by-bit ANDing of the accumulator with the contents of memory location Z, and

NOT

might imply the bit-by-bit complementation of the accumulator.

The *logical shift* operations are useful in manipulating logical data (and characters strings). During a logical shift, the bits are shifted left or right, and zeros are typically inserted into vacant positions. Sign copying is not performed as in arithmetic shifting.

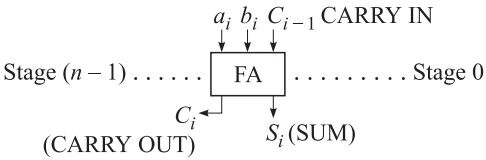
### Multiple-Bit Shift

ASC ALU is designed for one bit shift at a time. It is desirable to have the capability of performing multiple shifts using one instruction. Since the address field in the ASC instruction format for shift instructions is not used, it may represent the number of shifts. The control unit must be changed to recognize the shift count in the address field and perform the appropriate number of shifts.

## 10.1.2 Addition and Subtraction

The pseudo-parallel adder used in the ASC ALU is slow; the addition is complete only after the CARRY has rippled through the most significant bit. There are several techniques to increase the speed of addition. We will discuss one such technique (CARRY LOOK-AHEAD) below.

Consider the *i*th stage of the pseudo-parallel adder with inputs $a_i$, $b_i$, and $C_{i-1}$ (CARRY IN) and the outputs $S_i$ and $C_i$ (CARRY OUT) shown here:




We define two functions:

$$\text{Generate:} \quad G_i = a_i \cdot b_i. \tag{10.1}$$

$$\text{Propagate:} \quad P_i = a_i \oplus b_i. \tag{10.2}$$

These functions imply that the stage *i* generates a CARRY if $(a_i \cdot b_i) = 1$ and the CARRY $C_{i-1}$ is propagated to $C_i$ if $(a_i \oplus b_i) = 1$. Substituting $G_i$ and $P_i$

into the equations for the SUM and CARRY functions of the full adder, we get:

$$\begin{aligned} \text{SUM:}\quad S_i &= a_i \oplus b_i \oplus C_{i-1} \\ &= P_i \oplus C_{i-1} \end{aligned} \tag{10.3}$$

$$\begin{aligned} \text{CARRY:}\ C_i &= a_i \cdot b_i \cdot C'_{i-1} + a'_i \cdot b_i \cdot C_{i-1} + a_i \cdot b'_i \cdot C_{i-1} + a_i \cdot b_i \cdot C_{i-1} \\ &= a_i b_i + (a'_i b_i + a_i b'_i) C_{i-1} \\ &= a_i b_i + (a_i \oplus b_i) C_{i-1} \\ &= G_i + P_i C_{i-1} \end{aligned} \tag{10.4}$$

$G_i$ and $P_i$ can be generated simultaneously since $a_i$ and $b_i$ are available. Equation (10.3) implies that $S_i$ can be simultaneously generated if all CARRY IN ($C_{i-1}$) signals are available. From Eq. (10.4),

$$\begin{aligned} C_0 &= G_0 + P_0 C_{-1} \qquad (C_{-1} \text{ is CARRY IN to the right-most bit.}) \\ C_1 &= G_1 + C_0 P_1 \\ &= G_1 + G_0 P_1 + C_{-1} P_0 P_1 \\ &\ \ \vdots \\ C_i &= G_i + G_{i-1} P_i + G_{i-2} P_{i-1} P_i + \cdots \\ &\quad + G_0 P_1 P_2 \cdots P_i + C_{-1} P_0 P_1 \cdots P_i. \end{aligned} \tag{10.5}$$

From Eq. (10.5) it is seen that $C_i$ is a function of only $G$s and $P$s of the $i$th and earlier stages. All $C_i$s can be simultaneously generated because $G$s and $P$s are available. Figure 10.1 shows a 4-bit carry look-ahead adder (CLA) schematic and the detailed circuitry. This adder is faster than the ripple-carry adder since the delay is independent of the number of stages in the adder and equal to the delay of the three stages of circuitry in Fig. 10.1 (that is, about six gate delays). Four-bit and 8-bit off-the-shelf CLA units are available in TTL. It is possible to connect several such units to form larger adders.

Refer to Section 10.5 for details of an off-the-shelf ALU.

### 10.1.3 Multiplication

Multiplication can be performed by repeated addition of the multiplicand to itself multiplier number of times. In the binary system, multiplication by a power of 2 corresponds to shifting the multiplicand left by one position and hence multiplication can be performed by a series of shift and add operations.

---

**Example 10.1** Consider the multiplication of two four-bit numbers:

| | | | |
|---|---|---|---|
| Multiplicand ($D$) | 1011 × 1101 | Multiplier ($R$) | $D = d_{n-1}, \cdots, d_1 d_0$ |
| | 1011 | | |
| | 0000 | | $R = r_{n-1}, \cdots, r_1 r_0$ |
| | 1011 | Partial products | |
| | 1011 | | |
| | 10001111 | Product | |

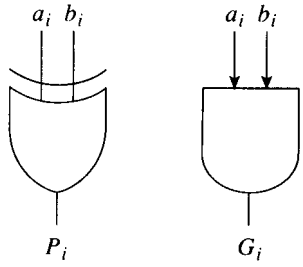
---

Note that each nonzero partial product above is the multiplicand shifted left an appropriate number of bits. Since the product of two $n$-bit numbers is $2n$ bits long, we can start off with a $2n$-bit accumulator containing all 0s and obtain the product by the following algorithm:

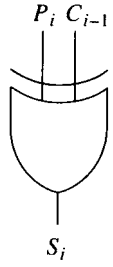
1. Perform the following step $n$ times ($i = 0$ through $n - 1$):
2. Test the multiplier bit $r_i$.
   If $r_i = 0$, shift accumulator right 1 bit;
   If $r_i = 1$, add multiplicand ($D$) to the most significant end of the accumulator and shift the accumulator right 1 bit.

Figure 10.2(a) shows a set of registers that can be used for multiplying two 4-bit numbers. In general, for multiplying two $n$-bit numbers, $R$ and $D$ registers are each $n$ bits long, and the accumulator ($A$) is $(n + 1)$ bits long. The concatenation of registers $A$ and $R$ (that is, $A \mathbin{¢} R$) will be used to store the product. The extra bit in $A$ can be used for representing the sign of the product. We will see from the multiplication example shown in Fig. 10.2(c) that the extra bit is needed to store the carry during partial product computation. The multiplication algorithm is shown in Fig. 10.2(b).

Various multiplication algorithms that perform multiplication faster than in the above example are available. Hardware multipliers are also available as off-the-shelf units. One such multiplier is described later in




(a) Propagate, CARRY generate




(b) SUM

(*continues*)

**Figure 10.1** CARRY LOOK-AHEAD adder (CLA)

Section 10.5. The book by Hwang listed as a reference at the end of this chapter provides details of multiplication algorithms and hardware.

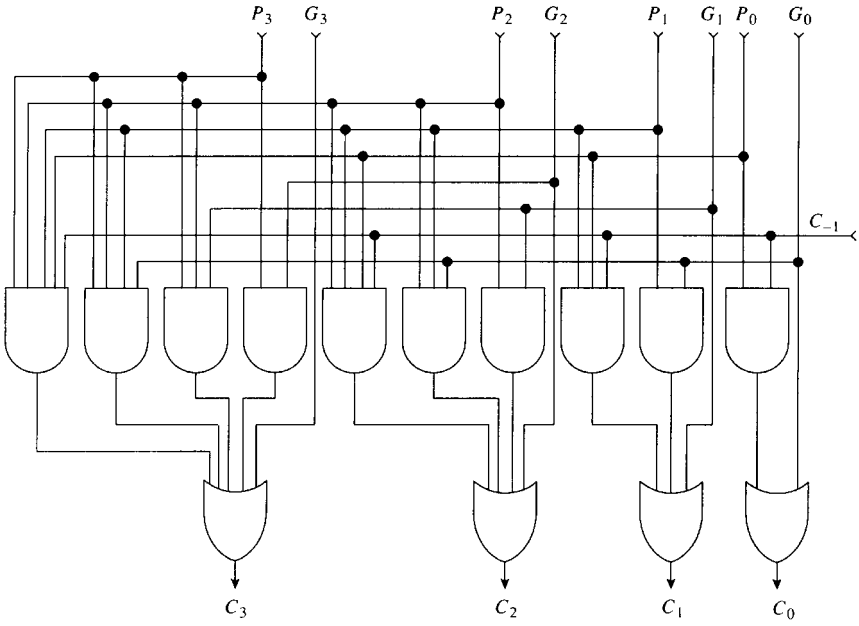
### 10.1.4 Division

Division can be performed by repeated subtraction. We will describe two division algorithms that utilize shift and add operations in this section. We will assume that an $n$-bit integer $X$ (dividend) is divided by another $n$-bit integer $Y$ (divisor) to obtain an $n$-bit quotient $Q$ and a remainder $R$, where

$$\frac{X}{Y} = Q + \frac{R}{Y} \tag{10.6}$$

and

$$0 \leq R < Y.$$




(c) CLA circuitry (4 bit)

(d) CLA schematic

**Figure 10.1** (Continued)

The first algorithm corresponds to the usual trial-and-error procedure for division and is illustrated with the following example. Let

$X = 1011$ (i.e., $n = 4$),
and $Y = 0011$,

$Q = q_3q_2q_1q_0$.

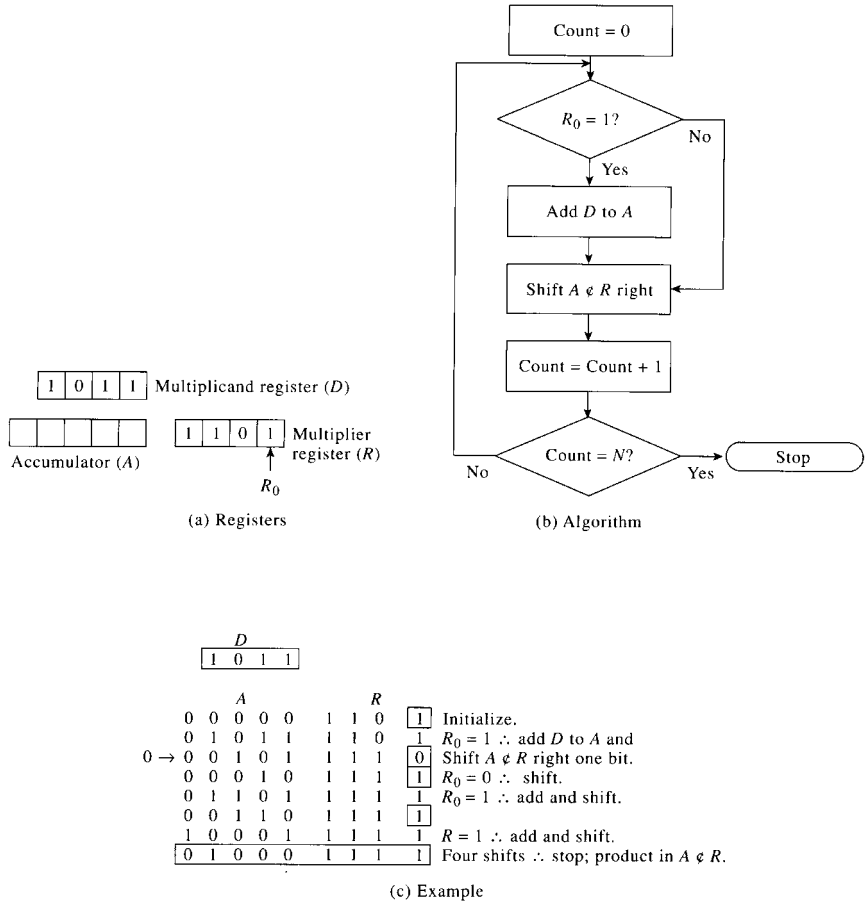

Count = 0
$R_0 = 1$?
No
Yes
Add $D$ to $A$
Shift $A$ ¢ $R$ right
Count = Count + 1
Count = $N$?
No
Yes
Stop
1 0 1 1 Multiplicand register ($D$)
Accumulator ($A$)
1 1 0 1 Multiplier register ($R$)
$R_0$
(a) Registers
(b) Algorithm
$D$
1 0 1 1
$A$ $R$
0 0 0 0 0 1 1 0 1 Initialize.
0 1 0 1 1 1 1 0 1 $R_0 = 1$ ∴ add $D$ to $A$ and
0 → 0 0 1 0 1 1 1 1 0 Shift $A$ ¢ $R$ right one bit.
0 0 0 1 0 1 1 1 1 $R_0 = 0$ ∴ shift.
0 1 1 0 1 1 1 1 1 $R_0 = 1$ ∴ add and shift.
0 0 1 1 0 1 1 1 1
1 0 0 0 1 1 1 1 1 $R = 1$ ∴ add and shift.
0 1 0 0 0 1 1 1 1 Four shifts ∴ stop; product in $A$ ¢ $R$.
(c) Example

**Figure 10.2** Multiplication

---

**Example 10.2**

| | |
|---|---|
| $0011\ \overline{)\ 000\ 1011}$ | Expand $X$ with $(n-1)$ zeros. |
| $-001\ 1$ | Subtract $2^{n-1}y$. |
| $-000\ 1101$ | Result is negative ∴ $q_{n-1} = 0$. |
| $+001\ 1$ | Restore by adding $2^{n-1}y$ back. |
| $000\ 1011$ | |
| $-\ 00\ 11$ | Subtract $2^{n-2}y$. |

| | |
|---|---|
| $-\ 000\ 0001$ | Result is negative $\therefore\ q_{n-2} = 0$. |
| $+\ \ 00\ 11$ | Restore by adding $2^{n-2}y$. |
| $000\ 1011$ | |
| $-\ \ 0\ 011$ | Subtract $2^{n-3}y$. |
| $000\ 0101$ | Result is positive $\therefore\ q_{n-3} = 1$. |
| $-\ 0011$ | Subtract $2^{n-4}y$. |
| $000\ 0010$ | Result is positive $\therefore\ q_{n-4} = 1$. |
| | Stop, after $n$ steps. |

$\therefore$ remainder = (0010), quotient = (0011).

(Note that in the subtractions peformed in this example, the two numbers are first compared, the smaller number is subtracted from the larger, and the result has the sign of the large number. For example, in the first step:

0001011 is smaller than 0011000;
hence, 0011000 – 0001011 = 0001101
and the result is negative)

---

This is called a *restoring division* algorithm, since the dividend is restored to its previous value if the result of the subtraction at any step is negative. If numbers are expressed in complement form, subtraction can be replaced by addition. The algorithms for an $n$-bit division are generalized below:

1. Assume the initial value of dividend $D$ is $(n-1)$ zeros concatenated with $X$.
2. Perform the following step for $i = (n-1)$ through 0: Subtract $2^i \cdot y$ from $D$. If the result is negative, $q_i = 0$ and restore $D$ by adding $2^i \cdot y$; if the result is positive, $q_i = 1$.
3. Collect $q_i$ to form the quotient $Q$; the value of $D$ after the last step is the remainder.

The second division algorithm is the *nonrestoring division* method. Example 10.3 illustrates this method.

---

**Example 10.3** Let $X = 1011$, $Y = 0110$.

| | |
|---|---|
| $0011\ \overline{)\ 000\ 1011}$ | Trial dividend. |
| $-011\ 0$ | Subtract $2^{n-1} \cdot Y$. |
| $-010\ 0101$ | Negative result $\therefore\ q_3 = 0$. |

| | |
|---|---|
| +01 10 | Add $2^{n-2}y \cdot Y$. |
| −000 1101 | Negative result $\therefore q_2 = 0$. |
| + 0 110 | Add $2^{n-3} \cdot Y$. |
| − 000 0001 | Negative result $\therefore q_1 = 0$. |
| + 0110 | Add $2^{n-4} \cdot Y$. |
| + 000 0101 | Positive result $\therefore q_0 = 1$. |

$\therefore$ quotient = 0001, remainder = 0101.

This method can be generalized into the following steps:

1. Assume initial value of dividend $D$ is $(n-1)$ zeros concatenated with $X$; set $i = n - 1$.
2. Subtract $2^{\text{i}} \cdot Y$ from $D$.
3. If the result is negative, $q_i = 0$; if the result is positive, $q_i = 1$, $i = i - 1$; go to step 4.
4. If the result is negative, add $2^i \cdot Y$ to the result; otherwise, subtract $2^i \cdot Y$ from the result to form new $D$; if $i = 0$, go to step 5; otherwise, go to step 3.
5. If the final result is negative, $q_0 = 0$; otherwise $q_0 = 1$. If the final result is negative, correct the remainder by adding $2^\circ \cdot Y$ to it.

The multiplication and division algorithms discussed above assume the operands are positive, and no provision is made for a sign bit, although the most significant bit of the result in division algorithms can be treated as a sign bit. Direct methods for multiplication and division of numbers represented in 1s and 2s complement forms are available.

The hardware implementation of multiply and divide is usually an optional feature on smaller machines. Multiply and divide algorithms can be implemented either in hardware or in firmware by developing microprograms to implement these algorithms or by using software routines, as in Chapter 4. Figure 10.3 shows the register usage for binary multiply-divide operations on the IBM 370.

### 10.1.5 Stack-based ALU

As described in Chapter 7, a zero-address machine uses the top two levels of the stack as operands for the majority of its operations. The two operands are discarded after the operation, and the result of the operation is pushed onto the stack.

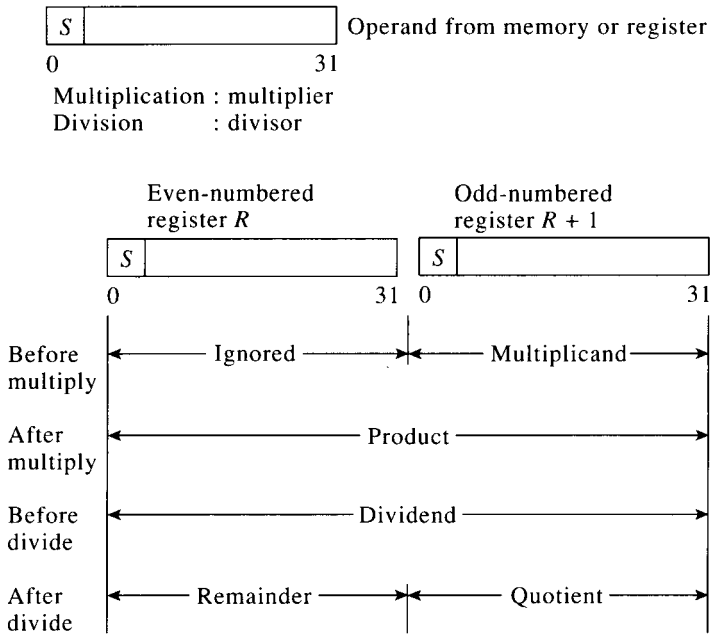



**Figure 10.3** Register usage for fixed-point multiply and divide on IBM 370

Stack-based ALU architectures have the following advantages:

1. The majority of operations are on the top two levels of the stack; hence, faster execution times are possible because address decoding and data fetch are not excessive.
2. Intermediate results in a computation usually can be left on the stack, thereby reducing the memory access needed and increasing the throughput.
3. Program lengths are reduced, since zero-address instructions are shorter compared to one-, two-, and three-address instructions.

Figure 10.4 shows the model of a stack. The data from the input register is pushed onto the stack when the PUSH control signal is activated. Contents of the top level of the stack are always available at the top-level output. POP when activated pops the stack, thereby moving the second level to the top level. The underflow error is generated when an attempt is made to pop the empty stack, and the overflow error condition occurs if an attempt is made to push the data into a full stack.

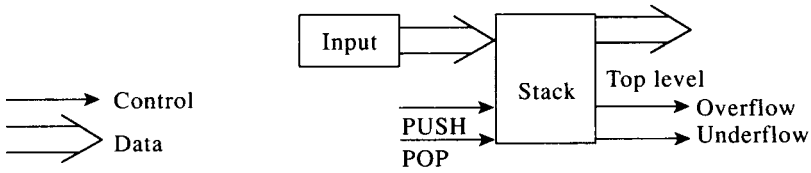
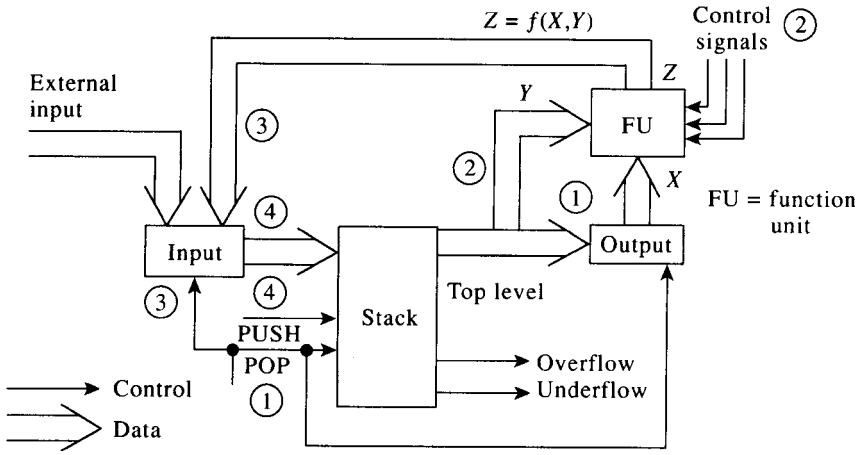



**Figure 10.4** Stack model

Appendix D provides the details of the two-popular implementations of stacks. The hardware (i.e., shift-register-based) implementation is used in ALU designs rather than the RAM-based implementation since fast stack operations are desired. Some machines use a combination of the shift-register- and RAM-based implementations in implementing stacks for the ALU. Here, a set of ALU registers are organized as the top few levels of the stack, and the subsequent levels are implemented in the RAM area. Thus, the levels needed most often can be accessed fast.

Figure 10.5 shows the details of a stack-based ALU. The functional unit is a combinational circuit that can perform operations such as addition, subtraction, etc. on its input operands $X$ and $Y$ and produce the result $Z$. The function of this module depends on the control signals from the control unit. In order to add two numbers on the top two levels of the stack, the following sequence of microoperations is needed:




Note: Numbers (①,②,...) indicate the sequence of operations.

**Figure 10.5** Stack ALU

| | Microoperation | Comments |
|---|---|---|
| ① | POP | OUTPUT ← Top level (i.e., operand $X$ is now available at the functional unit); Second and subsequent levels of the stack move up. The second operand is available on $Y$ input. |
| ② | ADD | Activate ADD control signal; the functional unit produces the sum of $X$ and $Y$ at its output $Z$. |
| ③ | POP | Pop the second operand from the stack and at the same time gate $Z$ into the input register. |
| ④ | PUSH | Push the result from the input register onto the stack |

The control unit should produce such microoperation sequences for each of the operations allowed in the ALU. Note in the above sequence that we asumed that the data is on the top two levels of the stack as a result of previous ALU operations.

## 10.2 DECIMAL ARITHMETIC

Some ALUs allow decimal arithmetic. In this mode, 4 bits are used to represent a decimal (BCD) digit and the arithmetic can be performed in two modes, *bit serial* and *digit serial*.

---

**Example 10.4** Figure 10.6 shows the addition of two decimal digits. In case 1, the sum results in a valid digit. In cases 2 and 3 the sum exceeds the highest valid digit 9, and hence a 6 is added to the sum to bring it to the proper decimal value. Thus the "ADD 6" correction is needed when the sum of the digits is between $(A)_{16}$ and $(F)_{16}$.

---

Figure 10.7(a) shows a digit serial (bit parallel) decimal adder. A bit serial adder is shown in (b). The bit serial adder is similar to the serial adder discussed in Chapter 2, except for the ADD 6 correction circuit. The sum bits enter the $B$ register from the left-hand input. At BIT 4, the decimal correction circuit examines sum bits $S_0$, $S_1$, $S_2$, and $S_3$ and returns the corrected sum to the $B$ register while generating an appropriate carry for the next digit.

Some processors use separate instructions for decimal arithmetic, while others use special instructions to switch the ALU between decimal and binary arithmetic modes. In the latter case, the same set of instructions operate both on decimal and binary data. M6502 uses a set decimal instruction to enter decimal mode and clear decimal instruction to return to binary

| | Case 1 | Case 2 | Case 3 | |
|---|---|---|---|---|
| | 5 | 6 | 7 | Decimal |
| | + 3 | 4 | 5 | |
| | 8 | *A* | *C*← | Hexadecimal |
| Correction | 0 | + 6 | + 6← | Decimal |
| | 8 | 10 | 12← | Decimal |

**Figure 10.6** BCD addition

mode. The arithmetic is performed in digit serial mode. Hewlett-Packard 35 system uses a serial arithmetic on 13-digit (52-bit) floating-point decimal numbers.

## 10.3 PIPELINING

As discussed in Chapter 9, in a pipelined architecture the instructions are executed in an overlapped manner. This mode of instruction processing was made possible by a multi-stage control unit design. Each stage performs its own set of operations on an instruction. The instruction is moved to the next stage for subsequent set of operations, while the current stage begins its processing on the next instruction.

---

**Example 10.5** The addition of two floating-point numbers is shown below:

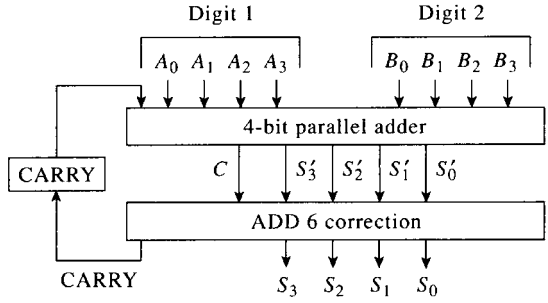
$(0.5 \times 10^{-3}) + (0.75 \times 10^{-2})$
$= 0.05 \times 10^{-2} + 0.75 \times 10^{-2}$ Equalize exponents.
$= 0.80 \times 10^{-2}$ Add mantissa.

If the mantissa of the sum is greater than 1, the exponent needs to be adjusted to make the mantissa less than 1 as in the following:

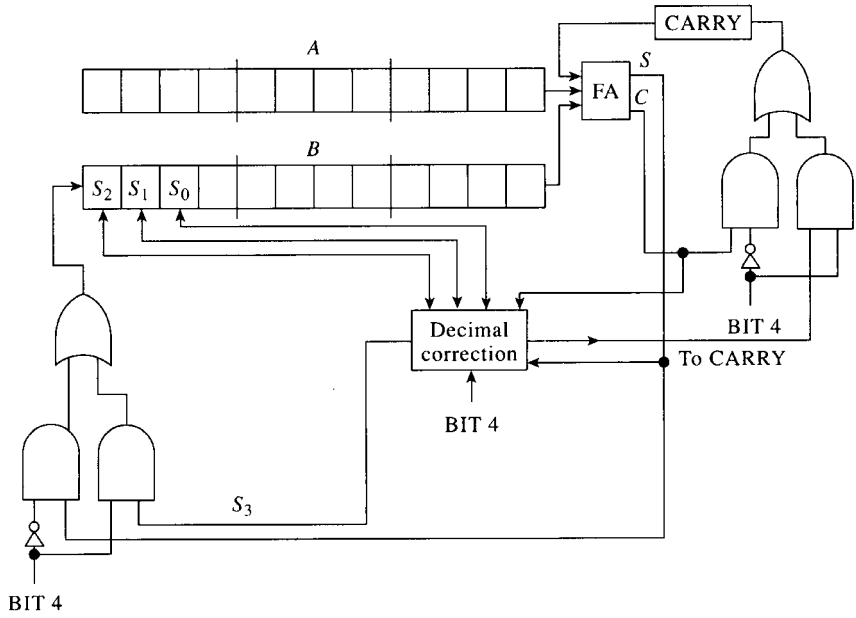
$0.5 \times 10^{-3} + 0.75 \times 10^{-3}$
$= 0.5 \times 10^{-3} + 0.75 \times 10^{-3}$ Equalize exponents.
$= 1.25 \times 10^{-3}$ Add mantissa.
$= 0.125 \times 10^{-2}$ Normalize mantissa.

---

These three steps involved in the floating-point addition can be used to design a three-stage pipeline as shown in Fig. 10.8. Each stage receives its




(a) Digit serial (bit parallel)




(b) Bit serial

Note: $S$ = SUM and $C$ = CARRY

**Figure 10.7** Decimal adder

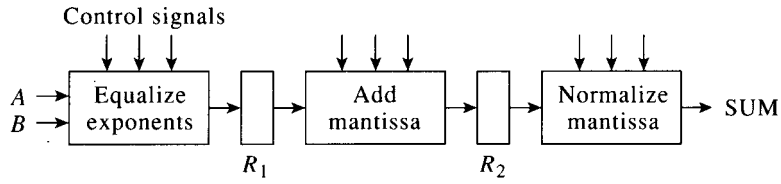


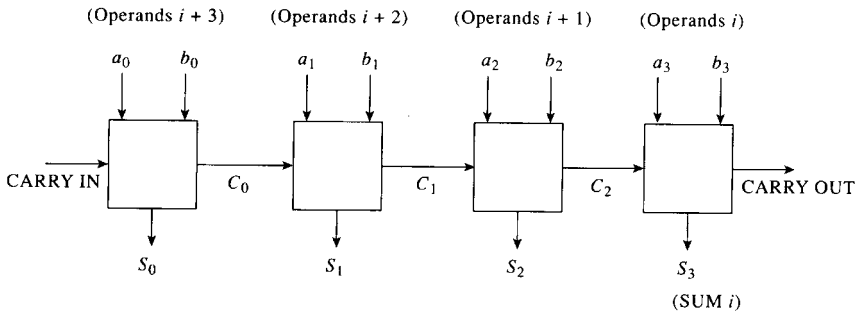

**Figure 10.8** Floating-point add pipeline

own control signals. $R_1$ and $R_2$ are holding registers to hold the data between stages. These buffer registers are especially needed when the processing times for stages are not equal. In addition to the data-holding registers, some flag bits to indicate the completion of processing may be needed at the output of each stage.

---

**Example 10.6** Figure 10.9 shows a pipeline scheme to add two 4-bit binary numbers. Here the carries propagate along the pipeline while the data bits to be added are input through the lateral inputs of the stages. Note that once the pipeline is full, each stage will be performing an addition. At any time, four sets of operands are being processed. The addition of the operands entering the pipeline at time $t$ will be complete at time $(t + 4\Delta)$ where $\Delta$ is the processing time of each stage. It is necessary to store the sum bits produced during four time slots to get the sum of a set of operands.

---




Note: $\text{SUM}_i$ complete at $i$, but the SUM bits were generated during $i$, $i-1$, $i-2$, and $i-3$.

**Figure 10.9** Floating-point binary adder

## 10.4 ALU WITH MULTIPLE FUNCTIONAL UNITS

An obvious method of increasing the throughput of the ALU is to employ multiple functional units in the design. Each functional unit can be either a general-purpose unit capable of performing all the ALU operations or a dedicated unit that can perform only certain operations. As many of these functional units as needed are activated by the control unit simultaneously, thereby achieving a high throughput. The control unit design becomes very complex since it is required to maintain the status of all the functional units and schedule the processing functions in an optimum manner, to take advantage of the possible parallelism.

We will now provide a brief description of two systems employing multiple functional units: the CDC-6600, and Cray X-MP in the next section.

## 10.5 EXAMPLE SYSTEMS

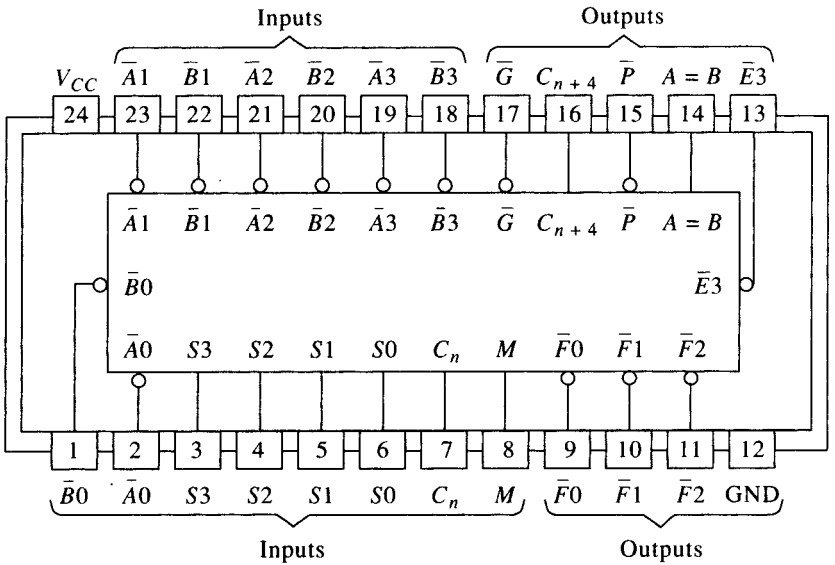
The Intel Pentium and the SGI MIPS series of processors described in earlier chapters are examples of CPUs with multiple processors. In this section, we provide brief descriptions of a number of other commercially available systems. The first example is an ALU IC capable of performing sixteen common arithmetic/logic functions. The second example is a hardware multiplier. The third example is a floating-point coprocessor, that supports an integer processor. The fourth example is an early system (CDC 6600) with multiple functional units, included here for its historical interest. The system formed the basis for the architecture of the Cray series of supercomputers; Cray X-MP system is described as the final example.

### 10.5.1 A Multifunction ALU IC (74181)

An ALU can be built using off-the-shelf ICs. For example, the SN 74181 shown in Fig. 10.10 is a multifunction ALU. It can perform sixteen binary arithmetic operations on two 4-bit operands $A$ and $B$. The function to be performed is selected by pins S0, S1, S2, and S3 and include addition, subtraction, decrement, and straight transfer, as shown in Fig. 10.10(b). $C_n$ is the CARRY INPUT and $C_{n+4}$ is the CARRY OUTPUT if the IC is used as a 4-bit ripple-carry adder. The IC can also be used with a look-ahead carry generator (74182) to build high-speed adders of multiple stages forming 8, 12, and 16-bit adders.

74181
Arithmetic logic units/
Function generators

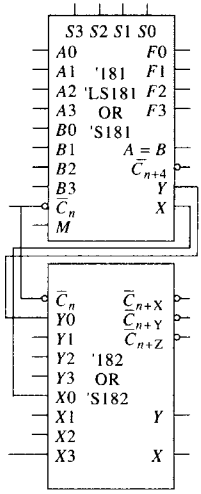
16 Arithmetic operations
16 Logic functions


Inputs
Outputs
$V_{CC}$ $\bar{A}1$ $\bar{B}1$ $\bar{A}2$ $\bar{B}2$ $\bar{A}3$ $\bar{B}3$ $\bar{G}$ $C_{n+4}$ $\bar{P}$ $A = B$ $\bar{E}3$
24 23 22 21 20 19 18 17 16 15 14 13
$\bar{A}1$ $\bar{B}1$ $\bar{A}2$ $\bar{B}2$ $\bar{A}3$ $\bar{B}3$ $\bar{G}$ $C_{n+4}$ $\bar{P}$ $A = B$
$\bar{B}0$
$\bar{E}3$
$\bar{A}0$ $S3$ $S2$ $S1$ $S0$ $C_n$ $M$ $\bar{F}0$ $\bar{F}1$ $\bar{F}2$
1 2 3 4 5 6 7 8 9 10 11 12
$\bar{B}0$ $\bar{A}0$ $S3$ $S2$ $S1$ $S0$ $C_n$ $M$ $\bar{F}0$ $\bar{F}1$ $\bar{F}2$ GND
Inputs
Outputs

(a) Pinout

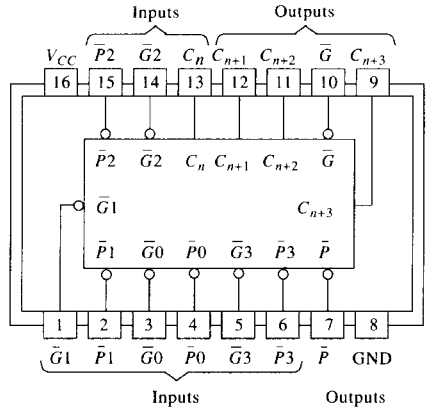
*(continues)*

**Figure 10.10** An ALU (Courtesy of Texas Instruments Incorporated.)




| Selection | Active-high data | | |
|---|---|---|---|
| | M = H | M = L: Arithmetic operations | |
| S3 S2 S1 S0 | Logic functions | $C_n$ = H (no carry) | $C_n$ = L (with carry) |
| L L L L | $F = \overline{A}$ | $F = A$ | $F = A$ plus 1 |
| L L L H | $F = \overline{A + B}$ | $F = A + B$ | $F = (A + \overline{B})$ plus 1 |
| L L H L | $F = \overline{A}B$ | $F = A + \overline{B}$ | $F = (A + \overline{B})$ plus 1 |
| L L H H | $F = 0$ | $F$ = minus 1 (2's COMPL) | $F$ = zero |
| L H L L | $F = \overline{A\overline{B}}$ | $F = A$ plus $A\overline{B}$ | $F = A$ plus $A\overline{B}$ plus 1 |
| L H L H | $F = \overline{B}$ | $F = (A + \overline{B})$ plus $A\overline{B}$ | $F + (A + \overline{B})$ plus $A\overline{B}$ plus 1 |
| L H H L | $F = A \oplus B$ | $F = A$ minus $B$ minus 1 | $F = A$ minus $B$ |
| L H H H | $F = A\overline{B}$ | $F = A\overline{B}$ minus 1 | $F = A\overline{B}$ |
| H L L L | $F = \overline{A} + B$ | $F = A$ plus $AB$ | $F = A$ plus $AB$ plus 1 |
| H L L H | $F = \overline{A \oplus B}$ | $F = A$ plus $B$ | $F = A$ plus $B$ plus 1 |
| H L H L | $F = B$ | $F$+ $(A + \overline{B})$ plus $AB$ | $F = (A + \overline{B})$ plus $AB$ plus 1 |
| H L H H | $F = AB$ | $F = AB$ minus 1 | $F = AB$ |
| H H L L | $F = 1$ | $F = A$ plus $A'$ | $F = A$ plus $A$ plus 1 |
| H H L H | $F = A + \overline{B}$ | $F = (A + B)$ plus $A$ | $F = (A + \overline{B})$ plus $A$ plus 1 |
| H H H L | $F = A + B$ | $F = (A + \overline{B})$ plus $A$ | $F = (A + \overline{B})$ plus $A$ plus 1 |
| H H H H | $F = A$ | $F = A$ minus 1 | $F = A$ |

(b) Four-bit with ALU carry look-ahead


Inputs
Outputs
$V_{CC}$ $\overline{P}2$ $\overline{G}2$ $C_n$ $C_{n+1}$ $C_{n+2}$ $\overline{G}$ $C_{n+3}$
16 15 14 13 12 11 10 9
$\overline{P}2$ $\overline{G}2$ $C_n$ $C_{n+1}$ $C_{n+2}$ $\overline{G}$
$\overline{G}1$ $C_{n+3}$
$\overline{P}1$ $\overline{G}0$ $\overline{P}0$ $\overline{G}3$ $\overline{P}3$ $\overline{P}$
1 2 3 4 5 6 7 8
$\overline{G}1$ $\overline{P}1$ $\overline{G}0$ $\overline{P}0$ $\overline{G}3$ $\overline{P}3$ $\overline{P}$ GND
Inputs
Outputs

(c) Look-ahead carry generator (74182)

**Figure 10.10** (Continued)

### 10.5.2 Texas Instruments' MSP 430 Hardware Multiplier

This section is extracted from: The MSP430 Hardware Multiplier: Functions and Applications, SLAA042, April 1999, Texas Instruments Inc.

The MSP 430 hardware multiplier allows three different multiply operations (modes):

- Multiplication of unsigned 8-bit and 16-bit operands (MPY)

- Multiplication of signed 8-bit and 16-bit operands (MPYS)
- Multiply-and-accumulate function (MAC) using unsigned 8-bit and 16-bit operands

Any mixture of operand lengths (8 and 16 bits) is possible. Additional operations are possible when supplemental software is used, such as signed multiply-and-accumulate.

Figure 10.11 shows the hardware modules that comprise the MSP430 multiplier. The accessible registers are explained in the following paragraphs. Figure 10.11 is not intended to depict the physical reality; it illustrates the hardware multiplier from the programmer's point of view. Figure 10.12 shows the system structure where the MSP430 is a peripheral.

The hardware multiplier is not part of the MSP430 CPU – it is a peripheral that does not interfere with the CPU activities. The multiplier uses normal peripheral registers that are loaded and read using CPU instructions. The programmer-accessible registers are explained next.

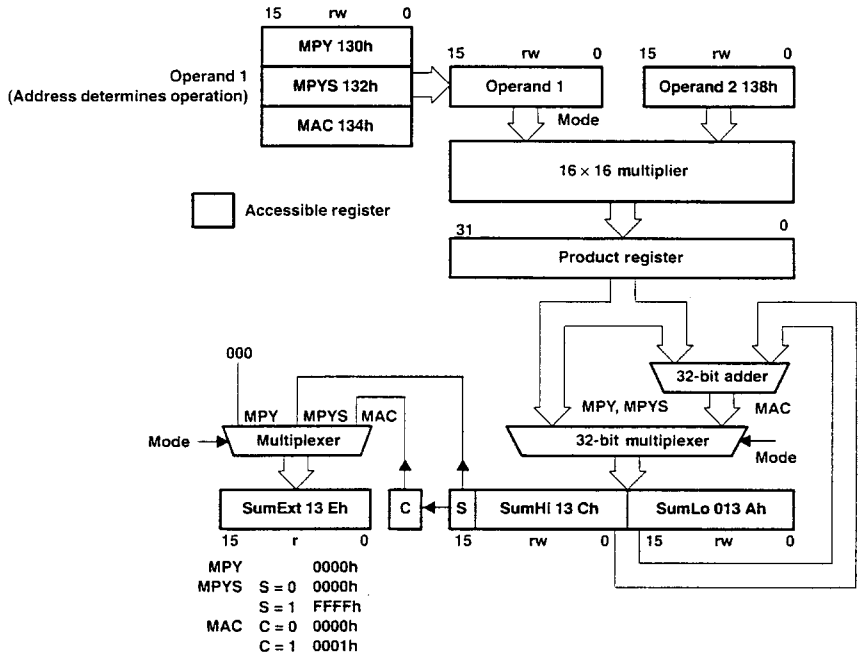



**Figure 10.11** Hardware multiplier block diagram

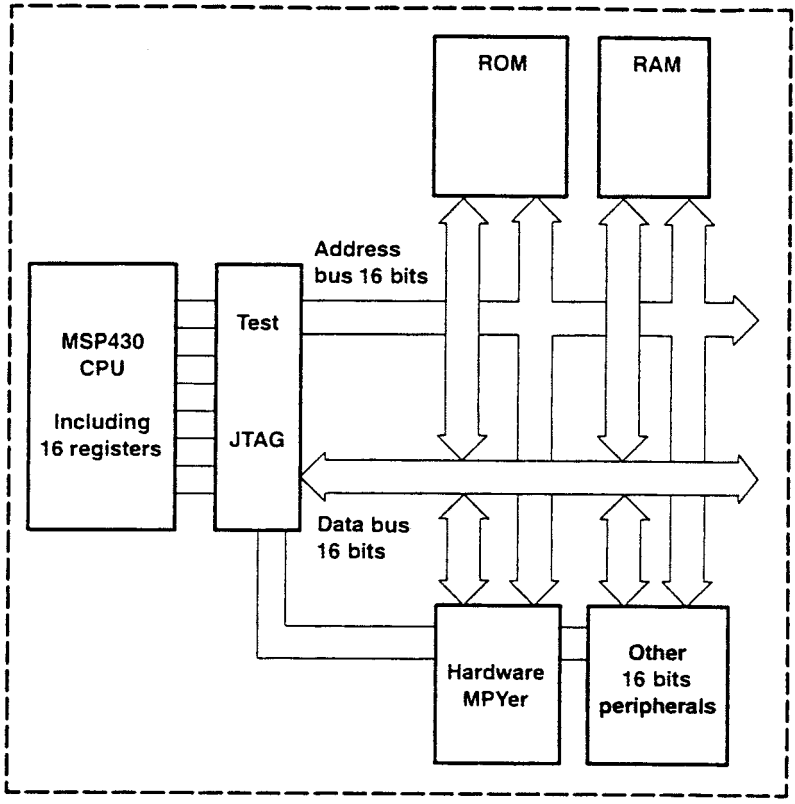



**Figure 10.12** Hardware multiplier internal connections

## Operand1 Registers

The operational mode of the MSP430 hardware multiplier is determined by the address where Operand1 is written:

- Address 130h: execute unsigned multiplication (MPY).
- Address 132h: execute signed multiplication (MPYS).
- Address 134h: execute unsigned multiply-and-accumulate (MAC).

The address of Operand1 alone determines the operation to be performed by the multiplier (after modification of Operand2). No operation is started by modification of Operand1 alone.

**Example 10.7** A multiply unsigned (MPY) operation is defined and started. The two operands reside in R14 and R15.

```
MOV  R15,&130h  ; Define MPY operation
MOV  R14,&138h  ; Start MPY with operand 2
. . .           ; Product in SumHi | SumLo
```

### Operand2 Register

The Operand2 Register (at address 138h) is common to all three multiplier modes. Modification of this register (normally with a MOV instruction) starts the selected multiplication of the two operands contained in Operand1 and Operand2 registers. The result is written immediately into the three hardware registers SumExt, SumHi, and SumLo. The result can be accessed with the next instruction, unless indirect addressing modes are used for source addressing.

### SumLo Register

This 16-bit register contains the lower 16 bits of the calculated product or sum. All instructions may be used to access or modify the SumLo register. The high byte cannot be accessed with byte instructions.

### SumHi Register

The contents of this 16-bit register, which depend on the previously executed operation, are as follows:

- MPY: the most-significant word of the calculated product.
- MPYS: the most-significant word, including the sign of the calculated product. Twos complement notation is used for the product.
- MAC: the most significant word of the calculated sum.

All instructions may be used with the SumHi register. The high byte cannot be accessed using byte instructions.

### SumExt Register

The sum extension register SumExt allows calculations with results exceeding the 32-bit range. This read-only register holds the most significant part of the result (bits 32 and higher). The content of SumExt is different for each multiplication mode:

- MPY: SumExt always contains zero, with no carry possible. The largest possible result is: 0FFFFh × 0FFFFh = 0FFFE0001h.
- MPYS: SumExt contains the extended sign of the 32-bit result (bit 31). If the result of the multiplication is negative (MSB = 1) SumExt contains 0FFFFh. If the result is positive (MSB = 0) SumExt contains 0000h.
- MAC: SumExt contains the carry of the accumulate operation. SumExt contains 0001 if a carry occurred during the accumulation of the new product to the old one, and zero otherwise.

Register SumExt simplifies multiple word operations – straightforward additions are performed without conditional jumps, saving time and ROM space.

---

**Example 10.8.** The new product of an MPYS operation (operands in R14 and R15) is added to a signed 64-bit result located in RAM words RESULT through RESULT + 6:

```
MOV    R15,&MPYS          ; First operand
MOV    R14,&OP2           ; Start MPYS with operand 2
ADD    SumLo, RESULT      ; Lower 16 bits of result
ADDC   SumHi, RESULT+2    ; Upper 16 bits
ADDC   SumExt, RESULT+4   ; Result bits 32 to 47
ADDC   SumExt, RESULT+6   ; Result bits 48 to 63
```

---

*Hardware Multiplier Rules.* The hardware multiplier is essentially a word module. The hardware registers can be addressed in word mode or in byte mode, but the byte mode can only address the lower bytes. The upper byte cannot be addressed.

The operand registers of the hardware multiplier (addresses 0130h, 0132h, 0134h and 0138h) behave like the CPU's working registers R0 to R15 when modified in byte mode: the upper byte is cleared in this case. This allows for any combination of 8-bit and 16-bit multiplications.

### Multiplication Modes

The three different multiplication modes available are explained in the following sections.

*Unsigned Multiply*. The two operands written to operand registers 1 and 2 are treated as unsigned numbers with:

| | |
|---|---|
| 00000h | being the smallest number |
| 0FFFFFh | being the largest number |

The maximum possible result is obtained with input operands 0FFFFh and 0FFFFh:

$$0FFFFh \times 0FFFFh = 0FFFE0001h$$

No carry is possible, the SumExt register always contains zero. Table 10.1 gives the products for some selected operands.

*Signed Multiply*. The two operands written to operand registers 1 and 2 are treated as signed twos complement numbers with:

| | |
|---|---|
| 08000h | being the most negative number (–32768) |
| 07FFFh | being the most positive number (+32767) |

The *SumExt* register contains the extended sign of the calculated result:

| | |
|---|---|
| SumExt = 00000h: | the result is positive |
| SumExt = 0FFFFh: | the result is negative |

Table 10.2 gives the signed-multiply products for some selected operands.

*Multiply-and-accumulate (MAC)*. The two operands written to operand registers 1 and 2 are treated as unsigned numbers (0h to 0FFFFh). The

**Table 10.1** Results with Unsigned-Multiply Mode

| Operands | SumExt | SumHi | SumLo |
|---|---|---|---|
| 0000 × 0000 | 0000 | 0000 | 0000 |
| 0001 × 0001 | 0000 | 0000 | 0001 |
| 7FFF × 7FFF | 0000 | 3FFF | 0001 |
| FFFF × FFFF | 0000 | FFFE | 0001 |
| 7FFF × FFFF | 0000 | 7FFF | 8001 |
| 8000 × 7FFF | 0000 | 3FFF | 8000 |
| 8000 × FFFF | 0000 | 7FFF | 8000 |
| 8000 × 8000 | 0000 | 4000 | 0000 |

**Table 10.2** Results with Signed-Multiply Mode

| Operands | SumExt | SumHi | SumLo |
|---|---|---|---|
| 0000 × 0000 | 0000 | 0000 | 0000 |
| 0001 × 0001 | 0000 | 0000 | 0001 |
| 7FFF × 7FFF | 0000 | 3FFF | 0001 |
| FFFF × FFFF | 0000 | 0000 | 0001 |
| 7FFF × FFFF | FFFF | FFFF | 8001 |
| 8000 × 7FFF | FFFF | C000 | 8000 |
| 8000 × FFFF | 0000 | 0000 | 8000 |
| 8000 × 8000 | 0000 | 4000 | 0000 |

maximum possible result is obtained with input operands 0FFFFh and 0FFFFh:

$$0FFFFh \times 0FFFFh = 0FFFE0001h$$

This result is added to the previous content of the two sum registers (SumLo and SumHi). If a carry occurs during this operation, the SumExt register contains 1 and is cleared otherwise.

SumExt = 00000h: no carry occurred during the accumulation
SumExt = 00001h: a carry occurred during the accumulation

The results of Table 10.3 assume that SumHi and SumLo contain the accumulated content C000,0000 before the execution of each example. See Table 10.1 for the results of an unsigned multiplication without accumulation.

*Multiplication word lengths.* The MSP430 hardware multiplier allows all possible combinations of 8-bit and 16-bit operands.

**Table 10.3** Results with Unsigned-Multiply-and-Accumulate Mode

| Operands | SumExt | SumHi | SumLo |
|---|---|---|---|
| 0000 × 0000 | 0000 | C000 | 0000 |
| 0001 × 0001 | 0000 | C000 | 0001 |
| 7FFF × 7FFF | 0000 | FFFF | 0001 |
| FFFF × FFFF | 0001 | BFFE | 0001 |
| 7FFF × FFFF | 0001 | 3FFE | 8001 |
| 8000 × 7FFF | 0000 | FFFF | 8000 |
| 8000 × FFFF | 0001 | 3FFF | 8000 |
| 8000 × 8000 | 0001 | 0000 | 0000 |

Notice that input registers Operand1 and Operand2 behave like CPU registers, where the high-register byte is cleared if the register is modified by a byte instruction. This simplifies the use of 8-bit operands. The following are examples of 8-bit operand use for all three modes of the hardware multiplier.

```
; Use the 8-bit operand in R5 for an unsigned multiply.
;   MOV.B   R5,&MPY      ; The high byte is cleared
;
; Use an 8-bit operand for a signed multiply.
    MOV.B   R5,&MPYS     ; The high byte is cleared
    SXT     &MPYS        ; Extend sign to high byte
;
; Use an 8-bit operand for a multiply-and-accumulate.
;   MOV.B   R5,&MAC      ; The high byte is cleared
```

Operand2 is loaded in a similar fashion. This allows all four possible combinations of input operands:

$16 \times 16$ $8 \times 16$ $16 \times 8$ $8 \times 8$

*Speed comparison with software multiplication.* Table 10.4 shows the speed increase for the four different $16 \times 16$-bit multiplication modes. The software loop cycles include the subroutine call (CALL #MULxx), the multiplication subroutine itself, and the RET instruction. Only CPU registers are used for the multiplication. See the *Metering Application Report* for details on the four multiplication subroutines.

The cycles given for the hardware multiplier include the loading of the multiplier operands Operand1 and Operand2 from CPU registers and – in the case of the signed MAC operation – the accumulation of the 48-bit result into three CPU registers.

Refer to the application report cited above for further details on programming the MSP430 and other application details.

**Table 10.4** CPU Cycles Needed with Different Multiplication Modes

| Operation | Software loop | Hardware MPYer | Speed increase |
|---|---|---|---|
| Unsigned multiply MPY | 139...171 | 8 | 17.4...21.4 |
| Unsigned MAC | 137...169 | 8 | 17.1...21.1 |
| Signed multiply MPYS | 145...179 | 8 | 18.1...22.4 |
| Signed MAC | 143...177 | 17 | 8.4...10.4 |

### 10.5.3 Motorola 68881 Coprocessor

The MC68881 floating-point coprocessor provides IEEE standard 754 floating-point operations capability as a logical extension to the integer data processing capabilities of MC68000 series of microprocessors (MPUs). The major features of the MC68881 are:

1. Eight general-purpose floating-point data registers
2. 67-bit arithmetic unit
3. 67-bit barrel shifter
4. Forty-six instructions
5. Full conformance to ANSI-IEEE 754-1985 standard
6. Supports additional functions not in IEEE standard
7. Seven data formats
8. Twenty-two constants available in on-chip ROM ($\pi$, e, etc.)
9. Virtual memory/machine operations
10. Efficient mechanisms for exception processing
11. Variable size data bus compatibility.

The coprocessor interface mechanism attempts to provide the programmer with an execution model based on sequential instruction execution in the MPU and the coprocessor. Floating-point instructions are executed concurrently with MPU integer instructions. The coprocessor and the MPU communicate via standard bus protocols.

Figure 10.13 shows the MC68881 simplified block diagram. The MC68881 is logically divided into two parts:

1. The bus interface unit (BIU)
2. The arithmetic processing unit (APU).

The task of the BIU is to communicate with the MPU. It monitors the state of the APU even though it operates independently of the APU. The BIU contains the coprocessor interface registers, communication status flags, and the timing control logic.

The task of the APU is to execute the command words and operands sent from the BIU. It must report its internal status to the BIU. The eight floating-point data registers, control, status, and instruction address registers are located in the APU. The high-speed arithmetic unit and the barrel shifter are also located in the APU. Figure 10.14 presents the programming model for the coprocessor.

The MC68881 floating-point coprocessor supports seven data formats:

1. Byte integer
2. Word integer
3. Long word integer

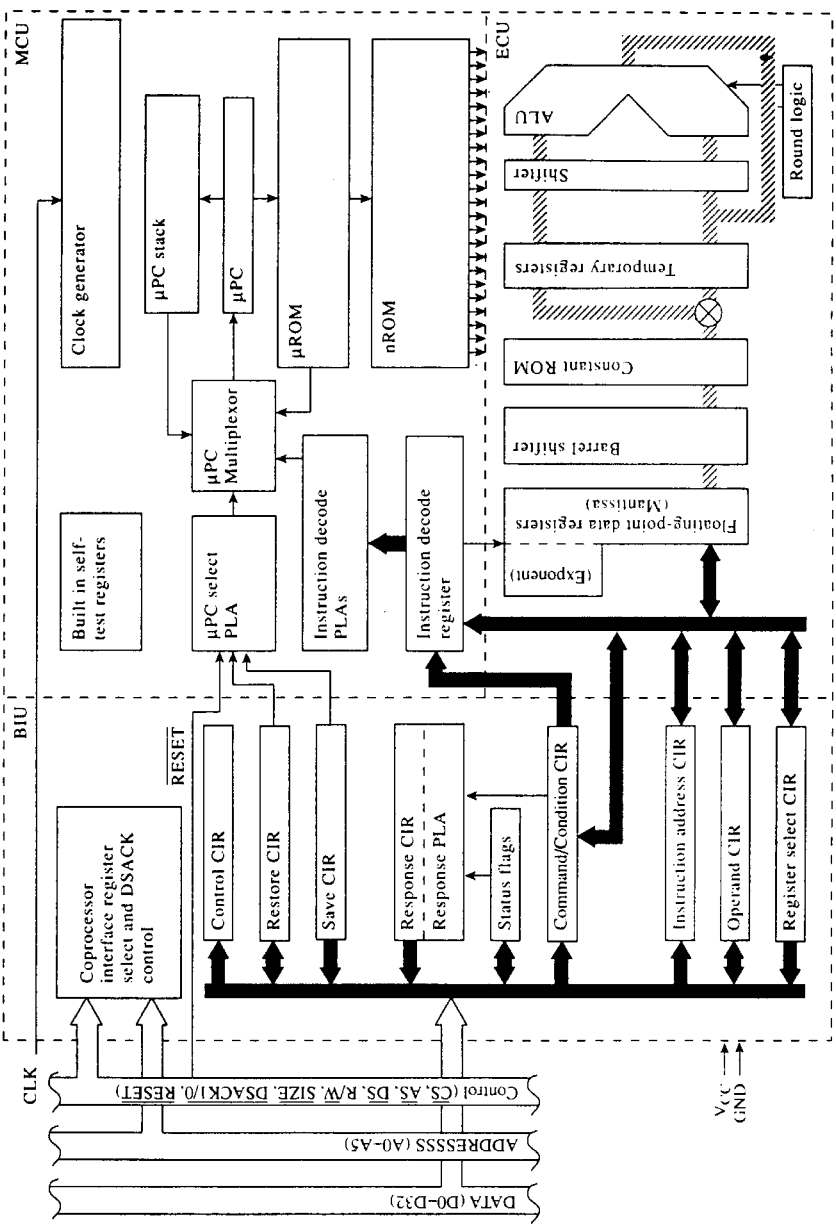



**Figure 10.13** MC68881 simplified block diagram (Courtesy of Motorola Inc.)

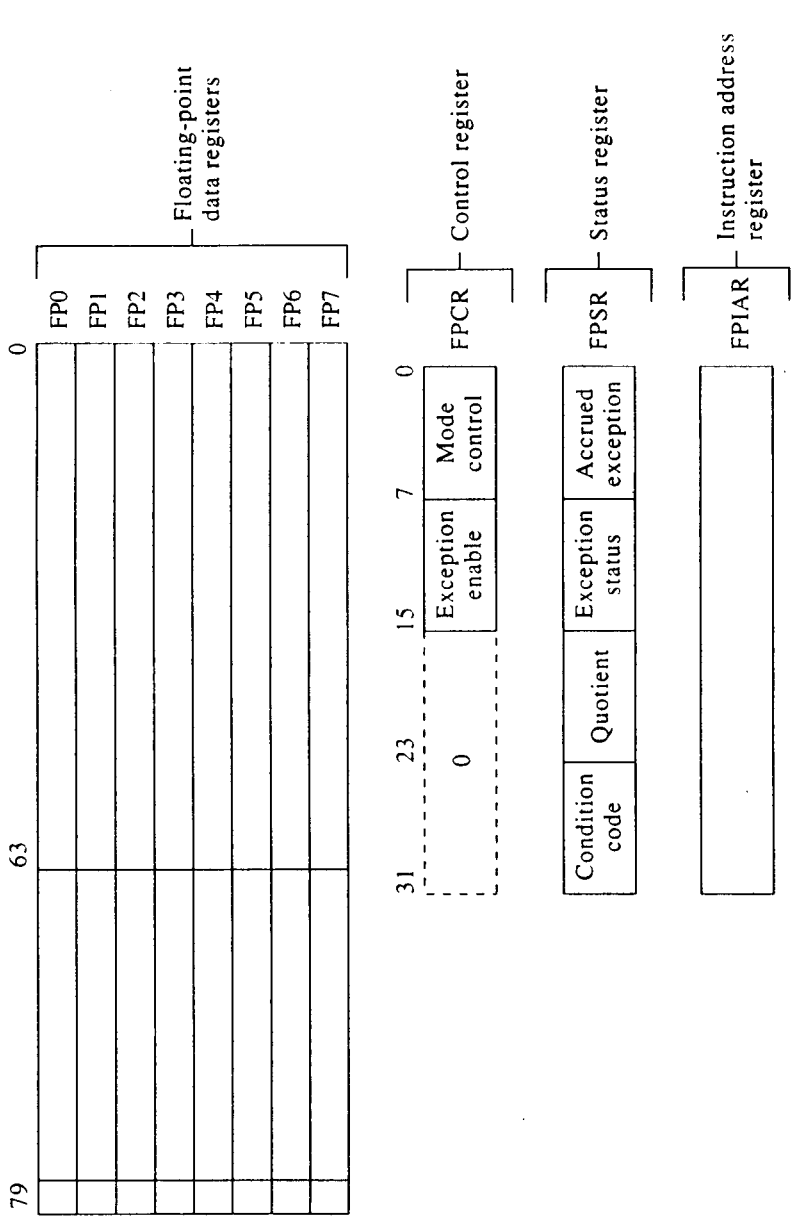



**Figure 10.14** MC68881 programming model (Courtesy of Motorola Inc.)

4. Single precision real
5. Double precision real
6. Extended precision real
7. Packed decimal string real.

The integer data formats consist of standard 2s complement format. Both single-precision and double-precision floating-point formats are implemented as specified in the IEEE standard. Mixed mode arithmetic is accomplished by converting integers to extended precision floating-point numbers before being used in the computation.

The MC68881 floating-point coprocessor provides six classes of instructions:

1. Moves between coprocessor and memory
2. Move multiple registers
3. Monadic operations
4. Dyadic operations
5. Branch, set, or trap conditionally
6. Miscellaneous.

These classes provide 46 instructions, which include 35 arithmetic operations.

### 10.5.4 Control Data 6600

The CDC 6600 first introduced in 1964 was designed for two types of use: large-scale scientific processing and time-sharing of smaller problems. To accommodate large-scale scientific processing, a high-speed, floating-point CPU employing multiple functional units was used. Figure 10.13 shows the structure of the system. As described in Chapter 6, the peripheral activity was separated from the CPU activity by using twelve I/O channels controlled by ten peripheral processors. Here, we will concentrate on the operation of the CPU.

As seen in Fig. 10.15 the CPU comprises ten functional units: one add, one double-precision add, two multiply, one divide, two increment, one shift, one boolean, and one branch units. The CPU obtains its programs and data from the central memory. It can be interrupted by a peripheral processor. The 10 functional units in the central processor can operate in parallel on one or two 60-bit operands to produce a 60-bit result. The operands and results are provided in the operating registers shown in Fig. 10.16. A functional unit is activated by the control unit as soon as the operands are available in the operating registers. Since the functional

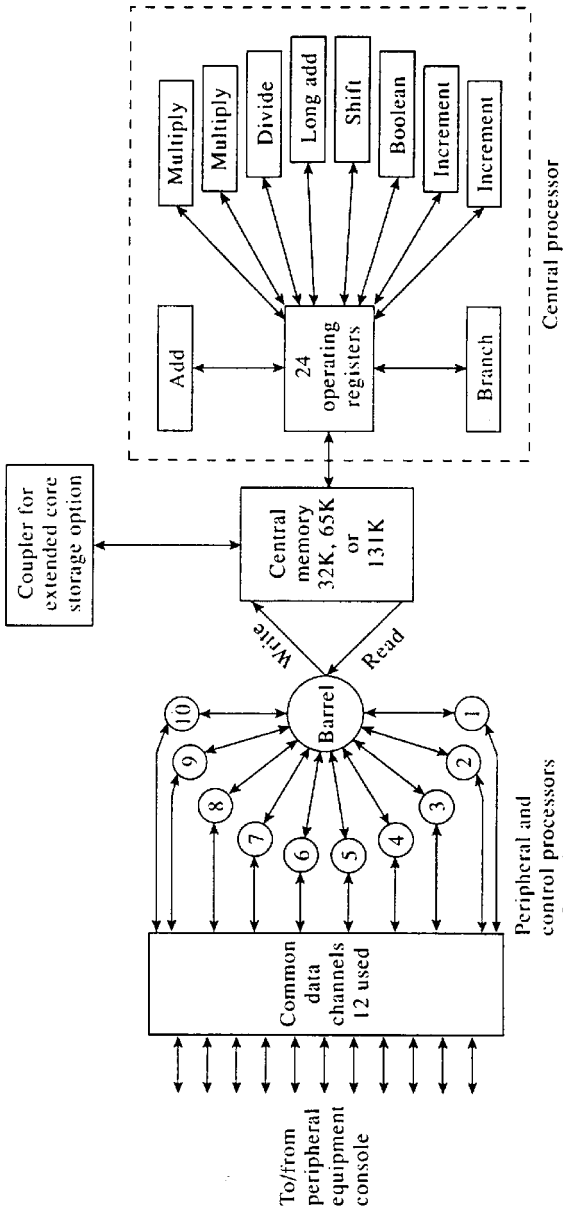



**Figure 10.15** CDC 6600 system structure (Courtesy of Control Data Corporation.)

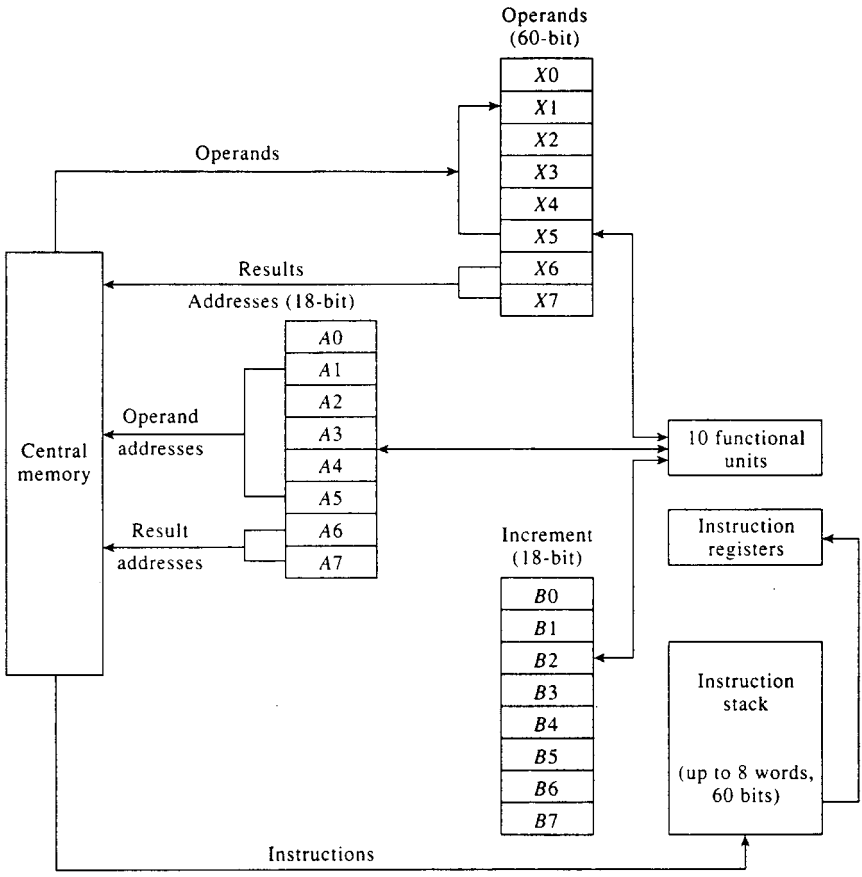



**Figure 10.16** Central processor operating registers of CDC 6600 (Courtesy of Control Data Corporation.)

units work concurrently, a number of arithmetic operations can be performed in parallel.

As an example, the computation of $Z = A \cdot B + C \cdot D$, where $Z$, $A$, $B$, $C$, and $D$ are memory operands, progresses as follows: First the contents of $A$, $B$, $C$, and $D$ are loaded into a set of CPU registers (says, $R1$, $R2$, $R3$, and $R4$ respectively). Then, $R1$ and $R2$ are assigned as inputs to a multiply unit with another register, $R5$, allocated to receive its output. Simultaneously, $R3$ and $R4$ are assigned as inputs to another multiply unit with $R6$ as its output. $R5$ and $R6$ are assigned as inputs to the add unit with $R7$ as its output. As soon as the multiply units provide their results into $R5$ and $R6$, they are

added by the add unit and the result from $R7$ is stored into $Z$. There is a queue associated with each functional unit. The CPU simply loads these queues with the operand and result register information. As and when a functional unit is free, it retrieves this information from its queue and operates on it, thus providing a very high parallelism.

There are 24 operating registers: eight 18-bit index registers, eight 18-bit address registers, and eight 60-bit floating-point registers. Figure 10.14 shows the data and address paths. Instructions are either 15 bits or 30 bits long. An instruction stack capable of holding 32 instructions enhances instruction execution speed (refer to Chapter 9).

The control unit maintains a *scoreboard*, which is a running file of the status of all registers and functional units and their allocation. As new instructions are fetched, resources are allocated to execute them by referring to the scoreboard. Instructions are queued for later processing if resources cannot be allocated.

Central memory is organized in 32 banks of 4K words. Each consecutive address calls for a different bank. Five memory trunks are provided between memory and five floating-point registers. An instruction calling for an address register implicitly initiates a memory reference on its trunk. An overlapped memory access and arithmetic operation is thus possible. The concurrent operation of functional units, high transfer rates between registers and memory, and separation of peripheral activity from the processing activity make CDC 6600 a fast machine. It should be noted that the CRAY supercomputer family designed by Seymour Cray, the designer of CDC 6600 series, retain the basic architectural features of the CDC 6600.

### 10.5.5 Architecture of the Cray Series

The Cray-1, a second-generation vector processor from Cray Research Inc. (now a subsidiary of Silicon Graphics Inc.) has been described as the most powerful computer of the late 1970s. Benchmark studies show that it is capable of sustaining computational rates of 138 MFLOPS over long periods of time and attaining speeds of up to 250 MFLOPS in short bursts. This performance is about five times that of the CDC 7600 or fifteen times that of a IBM System/370 Model 168. Thus Cray-1 is uniquely suited to the solution of computationally intensive problems encountered in fields such as weather forecasting, aircraft design, nuclear research, geophysical research, and seismic analysis.

Figure 10.17 shows the structure of the Cray X-MP (successor to Cray-1). A four-processor system (X-MP/4) is shown. The Cray X-MP consists of four sections: multiple Cray-1-like CPUs, the memory system,

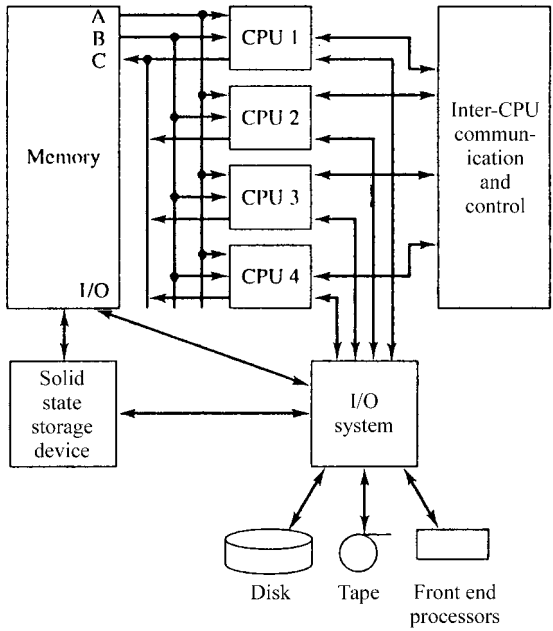



**Figure 10.17** Cray X-MP/4 structure

the I/O system and the processor interconnection. The following paragraphs provide a brief description of each section.

## Memory

The memory system is built out of several sections, each divided into banks. Addressing is interleaved across the sections and within sections across the banks. The total memory capacity can be up to 16 megawords with 64 bits/word. Associated with each memory word, there is an 8-bit field dedicated to single error correction/double error detection (SECDED). The memory system offers a bandwidth of 25–100 gigabits/second. It is multiported, with each CPU connected to four ports (two for reading, one for writing and one for independent I/O). Accessing a port ties it up for one clock cycle, and a bank access takes four clock cycles.

Memory contention can occur several ways: a bank conflict occurs when a bank is accessed while it is still processing a previous access; a simultaneous conflict occurs if a bank is referenced simultaneously on independent lines from different CPUs and a line conflict occurs when two or

more of the data paths make a memory request to the same memory section during the same clock cycle. Memory conflict resolution may require wait states to be inserted. Because memory conflict resolution occurs element by element during vector references, it is possible that the arithmetic pipelines being fed by these vector references may experience clock cycles with no input. This can produce a degradation in the arithmetic performance attained by the pipelined functional units. Memory performance is typically degraded by 3–7% on average due to memory contention, and in particularly bad cases by 20–33%.

The secondary memory, known as the solid state device (SSD), is used as an exceptionally fast-access disk device although it is built out of MOS random access memory ICs. The access time of SSD is 40 $\mu$s, compared to the 16 ms access time of the fastest disk drive from Cray Research Inc. The SSD is used for the storage of large-scale scientific programs that would otherwise exceed main memory capacity and to reduce bottlenecks in I/O-bound applications. The central memory is connected to the SSD through either one or two 1000 MB/s channels. The I/O subsystem is directly connected to the SSD, thereby allowing prefetching of large datasets from the disk system to the faster SSD.

### Processor Interconnection

The interconnection of the CPUs assumes a coarse-grained multiprocessing environment. That is, each processor (ideally) executes a task almost independently, requiring communication with other processors once every few million or billion instructions.

Processor interconnection comprises the clustered share registers. The processor may access any cluster that has been allocated to it in either user or system monitor mode. A processor in monitor mode has the processor ability to interrupt any other processor and cause it to go into monitor mode.

### Central Processor

Each CPU is composed of low-density ECL logic with 16 gates per chip. Wire lengths are minimized to cut the propagation delay of signals to about 650 ps. Each CPU is a register-oriented vector processor (Fig. 10.18) with various sets of registers that supply arguments to and receive results from several pipelined, independent functional units. There are eight 24-bit address registers (A0–A7), eight 64-bit scalar registers (S0–S7), and eight vector registers (V0–V7). Each vector register can hold up to sixty-four 64-

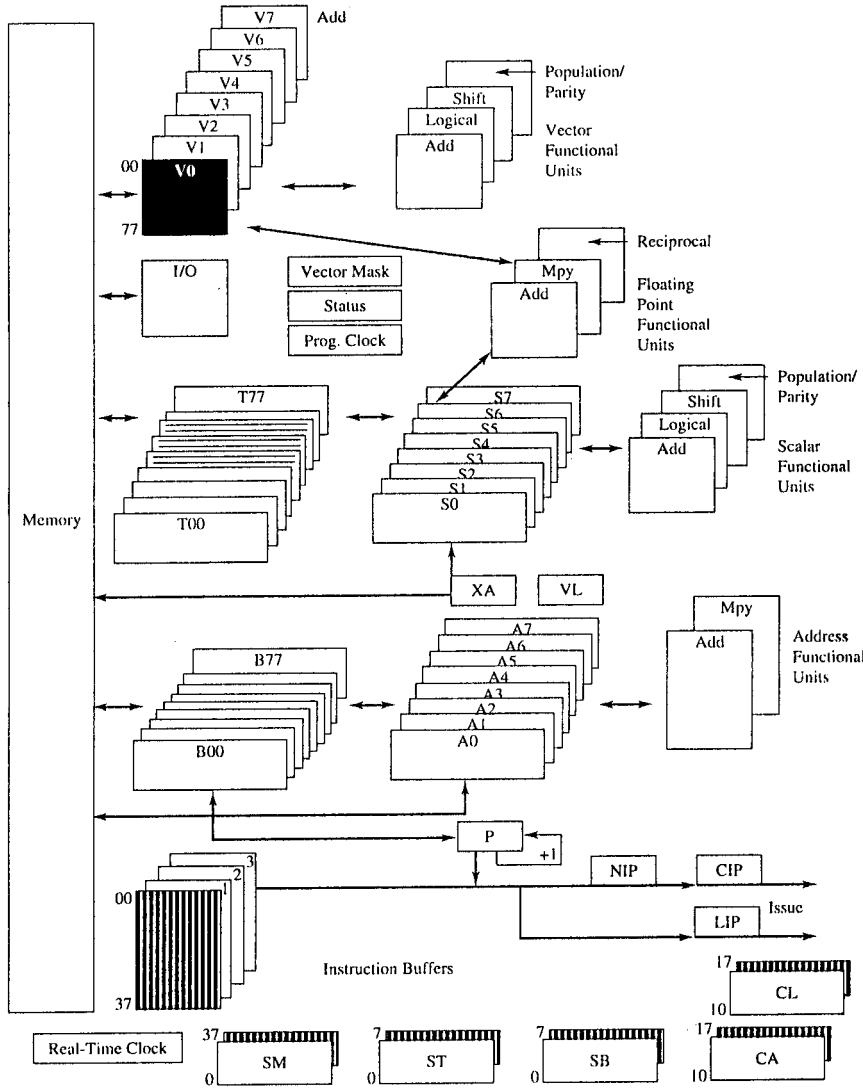



**Figure 10.18** Cray X-MP CPU. (Courtesy of Cray Research, Inc.)

bit elements. The number of elements present in a vector register for a given operation can be contained in a 7-bit vector length register (VL). A 64-bit vector mask register (VM) allows masking of the elements of a vector register prior to an operation. Sixty-four 24-bit address save registers (B0–B63) and sixty-four scalar save registers (T0–T63) are used as buffer storage areas for the address and scalar registers respectively.

The address registers support an integer add and an integer multiply functional unit. The scalar and vector registers each supports integer add, shift, logical, and population count functional units. Three floating-point arithmetic functional units (add, multiply, reciprocal approximation) take their arguments from either the vector or the scalar registers.

The result of the vector operation is either returned to another vector register or may replace one of the operands to the same functional unit (i.e., "written back") provided there is no recursion.

An 8-bit status register contains such flags as processor number, program status, cluster number, interrupt and error detection enables. This register can be accessed through an S register.

The exchange address register is an 8-bit register maintained by the operating system. This register is used to point to the current position of the exchange routine in memory. Also, there is program clock used for accurately measuring duration intervals.

As mentioned earlier, each CPU is provided with four ports to the memory with one port reserved for the input/output subsystem and the other three, labeled A, B, and C supporting data paths to the registers. All the data paths can be active simultaneously, as long as there are no memory access conflicts.

Data transfer between scalar and address registers and the memory occurs directly (i.e., as individual elements in to and out of referenced registers). Alternatively, block transfers can occur between the buffer registers and the memory. The transfer between scalar and address registers and the corresponding buffers is done directly. Transfers between the memory and the vector registers are done only directly.

Block transfer instructions are available for loading to and storing from B (using port A) and T (using port B) buffer registers. Block stores from the B and T registers to memory use port C. Loads and stores directly to the address and scalar registers use port C at a maximum data rate of one word every two clock cycles.

Transfers between the B and T registers and address and scalar registers occur at the rate of one word per clock cycle, and data can be moved between memory and the B and T registers at the same rate of one word per clock cycle. Using the three separate memory ports data can be moved beween common memory and the buffer registers at a combined rate of

three words per clock cycle, one word into B and T and one word from one of them.

The functional units are fully segmented (i.e., pipelined), which means that a new set of arguments may enter a unit every clock period (8.5 ns). The segmentation is performed by holding the operands entering the unit and the partial results at the end of every segment until a flag allowing them to proceed is set. The number of segments in a unit determines the start-up time for that unit. Table 10.5 shows the functional unit characteristics.

---

**Example 10.9** Consider again the vector addition:

$$C[i] = A[i] + B[i] \qquad 1 < i < N.$$

---

Assume that $N$ is 64, $A$ and $B$ are loaded in to two vector registers, and the result vector is stored in another vector register. The unit time for floating-point addition is six clock periods. Including one clock period for transfer-

**Table 10.5** Functional Unit Characteristics (Cray X-MP)

| Type | Operation | Registers used | Number of bits | Unit time (clock periods) |
|---|---|---|---|---|
| Address | Integer add | A | 24 | 2 |
| | Integer multiply | A | 24 | 4 |
| Scalar | Integer add | S | 64 | 3 |
| | Shift | S | 64 | 2 or 3 |
| | Logical | S | 64 | 1 |
| | Population | S | 64 | 3 or 4 |
| | Parity, and Leading zero | S & A | 64 | 3 or 4 |
| Vector | Integer add | V | 64 | 3 |
| | Shift | V & A | 64 | 3 or 4 |
| | Logical | V | 64 | 2 |
| | Second logical | V | 64 | 3 |
| | Population, and parity | V | 64 | 6 |
| Floating point | Add | S or V | 64 | |
| | Multiply | S or V | 64 | |
| | Reciprocal | S or V | 64 | |
| Memory transfer | Scalar load | S | 64 | |
| | Vector load (64 element) | V | 64 | |

ring data from vector registers to the add unit and one clock to store the result into another vector register, it would take ($64 \times 8 = 512$) clock periods to execute this addition in scalar mode.

This vector operation performed in the pipeline mode is shown in Fig. 10.19. Here, the first element of the result will be stored into the vector register after 8 clock periods. Afterwards there will be one result every clock period. Therefore the total execution time is ($8 + 63 = 71$) clock periods.

If $N < 64$, the above execution times are reduced correspondingly. If $N > 64$, the computation is performed on units of 64 elements at a time. For example if $N$ is 300, the computation is performed on four sets of 64 elements each, followed by the final set with the remaining 44 elements.

The vector length register contains the number of elements ($n$) to be operated upon at each computation. If $M$ is the length of vector registers in the machine, the following program can be used to execute the above vector operation for an arbitrary value of $N$.

```
begin = 1
n = (N mod M)
for i = 0, (N/M)
   for j = begin, begin + n − 1
   C[j] = A[j] + B[j]
   endfor
begin = begin + n
n = M.
endfor
```

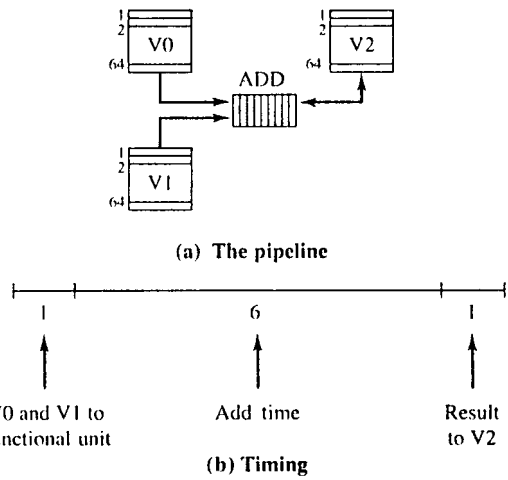



**Figure 10.19** Vector pipelining on Cray X-MP.

Here, first the ($N$ mod $M$) elements are operated upon, followed by $N/M$ sets of $M$ elements.

In practice, the vector length will not be known at compile time. The compiler generates the code similar to above, such that the vector operation is performed in sets of length less than or equal to $M$. This is known as *strip mining*. Strip mining overhead must also be included in start-up time computations of the pipeline.

In order for multiple functional units to be active simultaneously, intermediate results must be stored in the CPU registers. When properly programmed, the Cray architecture can arrange CPU registers such that the results of one functional unit can be input to another independent functional unit. Thus, in addition to the pipelining within the functional units, it is possible to pipeline arithmetic operations between the functional units. This is called *chaining*.

Chaining of vector functional units and their overlapped, concurrent operation are important characteristics of this architecture, which bring about a vast speed-up in the execution times. The following example shows a loop where overlapping would occur.

---

**Example 10.10** Consider the loop:

For $J = 1,64$
$C(J) = A(J) + B(J)$
$D(J) = E(J)^*F(J)$
Endfor

Here, the addition and multiplication can be done in parallel because the functional units are totally independent.

---

The following example illustrates chaining.

---

**Example 10.11**

For $J = 1,64$
$C(J) = A(J) + B(J)$
$D(J) = C(J)^*E(J)$
Endfor

Here, the output of the add functional unit is an input operand to the multiplication functional unit. With chaining, we do not have to wait for the entire array $C$ to be computed before beginning the multiplication. As

soon as $C(1)$ is computed, it can be used by the multiply functional unit concurrently with the computation of $C(2)$. That is, the two functional units form the stages of a pipeline as shown in Fig. 10.20.

---

Assuming that all the operands are in vector registers, this computation is done without vectorization (i.e., no pipelining or chaining), requires (add: 64*8 = 512 plus multiply: 64*9 = 576) 1088 clock periods. It can be completed in (chain start-up time of 17 plus 63 more computations) 80 clock periods if vectorization (pipelining and chaining) is employed.

Note that the effect of chaining is to increase the depth of the pipeline and hence the start-up overheads. If the operands are not already in vector registers, they need to be loaded first and the result stored in the memory. The two load paths and the path that stores data to memory can be considered as functional units in chaining. The start-up time for a load vector instruction is 17 cycles, and thereafter one value per cycle may be fetched, then any operation using this data may access one value per cycle after 18 cycles. Figure 10.21 shows an example of this. Here, Port A is used to read in V0 and port B to read in V1. This occurs in parallel. As soon as each vector register has its first operand, the floating-point add may begin pro-

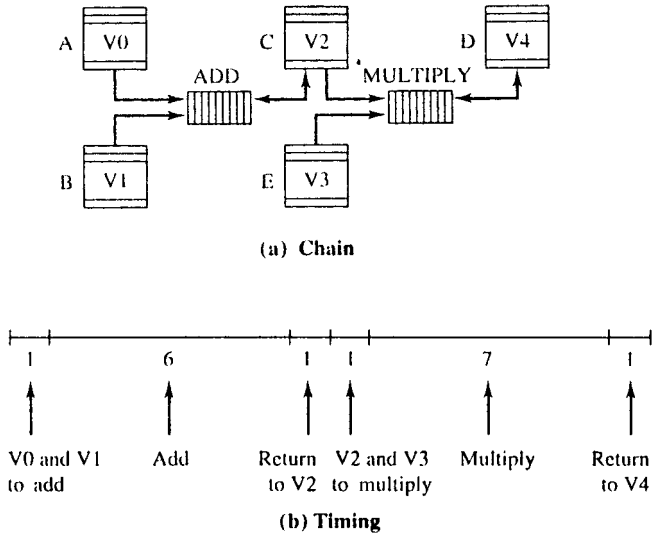



**Figure 10.20** Chaining in Cray X-MP

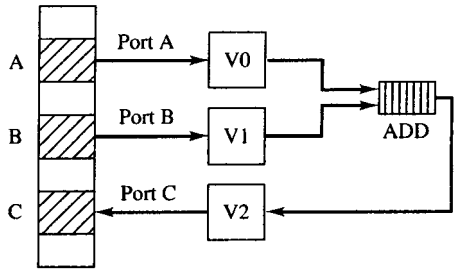



**Figure 10.21** Memory and functional unit chaining on Cray X-MP

cessing, and as soon as the first operand is placed in V2, port C may be used to store it back to memory.

In a chain, a functional unit can only appear once. Two fetches and one store are possible in each chain. This is because Cray systems supply only one of each of the above types of functional units. This demands that if two floating-point adds are to be executed, they must occur sequentially. Because there are two ports for fetching vectors and one port for storing vectors, the user may view the system as having two load functional units and a store functional unit on Cray X-MP. On Cray-1, there is only one memory functional unit.

An operand can only serve as input to one arithmetic functional unit in a chain. An operand can, however, be input to both inputs of a functional unit requiring two operands. This is because a vector register is tied to a functional unit during a vector instruction. When a vector instruction is issued, the functional unit and registers used in the instruction are reserved for the duration of the vector instruction.

Cray Research has coined the term 'chime' (chained vector time) to describe the timing of vector operations. A chime is not a specific amount of time, but rather a timing concept representing the number of clock periods required to complete one vector operation. This equates to length of a vector register plus a few clock periods for chaining. For Cray systems, a chime is equal to 64 clock periods plus a few more. A chime is thus a measure that allows the user to estimate the speed-up available from pipelining, chaining and overlapping instructions.

The number of chimes needed to complete a sequence of vector instructions is dependent on several factors. Since there are three memory functional units, two fetches and one store operation may appear in the same chime. A functional unit may be used only once within a chime. An operand may appear as input to only one functional unit in a chime. A store operation may chain onto any previous operation.

**Example 10.12** Figure 10.22 shows the number of chimes required to perform the following code on Cray X-MP, Cray-1, and the Cray-2 [Levesque and Williamson, 1989] systems:

For $I = 1$ to 64
$A(I) = 3.0^{*}A(I) + (2.0 + B(I))^{*}C(I)$
Endfor

The Cray X-MP requires only two chimes, while the Cray-1 requires four, and the Cray-2, which does not allow chaining, requires six chimes to execute the code. Clock cycles of the Cray-1, the Cray X-MP and the Cray-2 are 12.5, 8.5, and 4.1 ns, respectively. If a chime is taken to be 64 clock cycles, then the time for each machine to complete the code is:

Cray-1: 4 chimes * 64 * 12.5 ns = 3200 ns
Cray X-MP: 2 chimes * 64 * 8.5 ns = 1088 ns
Cray-2: 6 chimes * 64 * 4.1 ns = 1574 ns

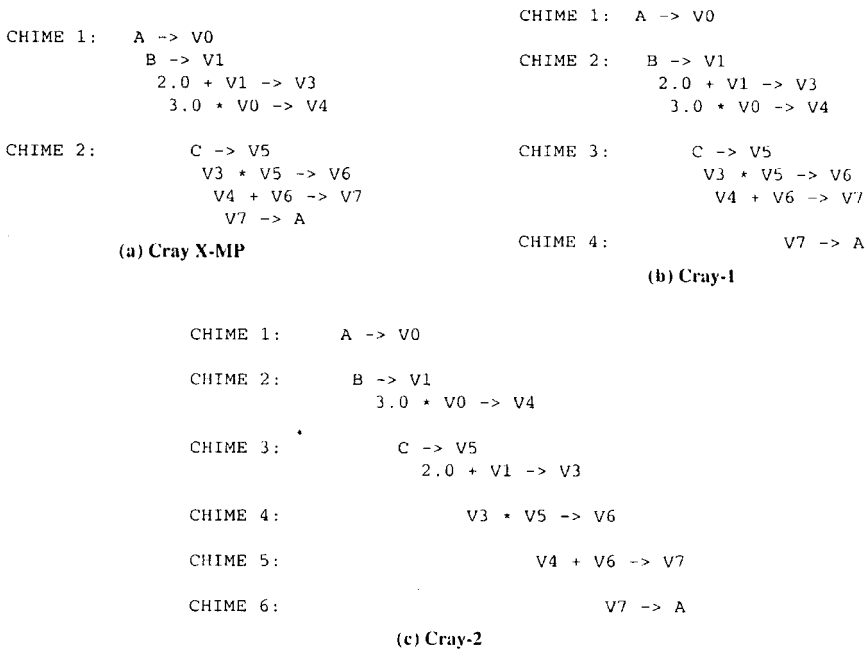

CHIME 1: A -> V0
B -> V1
2.0 + V1 -> V3
3.0 * V0 -> V4
CHIME 2: C -> V5
V3 * V5 -> V6
V4 + V6 -> V7
V7 -> A
(a) Cray X-MP
CHIME 1: A -> V0
CHIME 2: B -> V1
2.0 + V1 -> V3
3.0 * V0 -> V4
CHIME 3: C -> V5
V3 * V5 -> V6
V4 + V6 -> V7
CHIME 4: V7 -> A
(b) Cray-1
CHIME 1: A -> V0
CHIME 2: B -> V1
3.0 * V0 -> V4
CHIME 3: C -> V5
2.0 + V1 -> V3
CHIME 4: V3 * V5 -> V6
CHIME 5: V4 + V6 -> V7
CHIME 6: V7 -> A
(c) Cray-2

**Figure 10.22** Chime characteristics

Thus, for some instruction sequences, the Cray X-MP with the help of chaining can actually outperform Cray-2, which has a faster clock. Since Cray-2 does allow overlapping, the actual gain of Cray X-MP may not be as large for very large array dimensions.

---

During vector operations, up to 64 target addresses could be generated by one instruction. If a cache were to be used as intermediate memory, the overhead to search for 64 addresses would be prohibitive. Use of individually addressed registers eliminates this overhead. One disadvantage of not using a cache is that the programmer (or the compiler) must generate all the references to the individual registers. This adds to the complexity of code (or compiler) development.

### Instruction Fetch

Control registers are part of the special purpose register set and are used to control the flow of instructions. Four instruction buffers, each containing 32 words (128 parcels, 16 bits each) are used to hold instructions fetched from memory. Each instruction buffer has its own instruction buffer address register (IBAR). The IBAR serves to indicate what instructions are currently in the buffer. The contents of IBAR are the high-order 17 bits of the words in the buffer. The instruction buffer is always loaded on a 32-word boundary. The P register is the 24-bit address of the next parcel to be executed. The current instruction parcel (CIP) contains the instruction waiting to issue and the next instruction parcel (NIP) contains the next instruction parcel to issue after the parcel in the CIP. Also, there is a last instruction parcel (LIP) which is used to provide the second parcel to a 32-bit instruction without using an extra clock period.

The P register contains the address of the next instruction to be decoded. Each buffer is checked to see if the instruction is located in the buffers. If the address is found the instruction sequence continues. However, if the address is not found, the instruction must be fetched from memory after the parcels in the CIP and NIP have been issued. The least recently filled buffer is selected to be overwritten, so that the current instruction is among the first eight words to be read. The rest of the buffer is then filled in a circular fashion until the buffer is full. It will take three clock pulses to complete the filling of the buffer. Any branch to an out-of-buffer address causes a delay in processing of 16 clock pulses.

Some buffers are shared between all of the processors in the system. One of these is the real-time clock. Other registers of this type include a

cluster consisting of 48 registers. Each cluster contains 32 (1-bit) semaphore or synchronization registers and eight 64-bit ST or shared-T registers and eight 24-bit SB or shared-B registers. A system with two processors will contain three clusters while a four-processor system will contain five clusters.

### I/O System

The input and output of the X-MP is handled by the input/output subsystem (IOS). The IOS is made of two to four interconnected I/O processors. The IOS receives data from four 100 MB/s channels connected directly to the main memory of the X-MP. Also, four 6 MB/s channels are provided to furnish control between the CPU and the IOS. Each processor has its own local memory and shares a common buffer. The IOS supports a variety of front-end processors and peripherals such as disk drives and drives.

To support the IOS, each CPU has two types of I/O control registers: current address and channel limit registers. The current address registers point to the current word being transferred. The channel limit registers contain the address of the last word to be transferred.

### Other Systems in the Series

Cray Research continued the enhancement of X-MP architecture to Y-MP and Y-prime series, while the enhancement of the Cray-2 line had been taken over by Cray Computer Corporation to develop Cray-3 and Cray-4 architectures.

Cray Y-MP was introduced in February 1988. It is of the same genre as the Cray X-MP. Cray-3 is a progression of Cray-2 architecture and is the first major system built with gallium arsenide (GaAs) technology. GaAs provides switching speeds of about five times that of silicon semiconductor technology, can handle higher frequencies, generates less noise, is more resistant to radiation, and can operate over a wider temperature range. With its 2 ns clock rate, the Cray-3 is ten times faster and less than half of the size of its predecessor the Cray-2. A brief description of these two architectures is provided below as a representation of the apparent dichotomy of design strategies being employed.

The Y-MP is designed to appear as an X-MP to the user. It extends the X-MP's 24-bit addressing scheme to 32 bits, thus allowing an address space of 32 million 64-bit words. It runs on a 6 ns clock and uses eight processors, thus doubling the processing speed.

The Cray-3 is a 16-processor machine, with a 2–2.5 ns clock rate. The projected speed is 16 GFLOPS and the projected memory capacity is 512 million words.

The Y-prime series utilizes silicon VLSI to improve on Y-MP while the Cray-4, a GaAs machine, would use 64 processors and 1 ns clock rate to achieve a speed of 128 GFLOPS.

## 10.6 SUMMARY

The spectrum of enhancements possible for ALUs spanning the range of employing faster hardware and algorithms to multiple functional units were described in this chapter. Several algorithms have been proposed over the years to increase the speed of the ALU functions. Only representative algorithms were described in this chapter. The reader is referred to the books listed in the References section for further details. The advances in hardware technology have resulted in several fast, off-the-shelf ALU building blocks. Some examples of such ICs were given. Examples of pipelined ALU architectures were provided. The trend in ALU design is to employ multiple processing units and activate them in parallel so that very high speeds can be achieved. In microcomputer-based system design, it is now common to utilize specialized coprocessors in parallel with the main processor. Several example systems were also described in this chapter.

## REFERENCES

August, M. C., *et al.* "Cray X-MP: The Birth of a Supercomputer," *IEEE Computer*, pp. 45–52, January 1989.

Cheung, T., and Smith, J. E., "An Analysis of the Cray X-MP Memory System", *Proc. IEEE Int. Conf. Parallel Processing*, pp. 499–505, 1984.

Cray Research, Inc., *Cray X-MP and Cray Y-MP computer systems* (Training Workbook), 1988.

Culler, D. E., Singh, J. and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA: Morgan Kaufmann, 1998.

Hill, M. D., Hockney, R. W., Jesshope, C. R., *Parallel Computers 2*, Philadelphia, PA: Adam Hilger, 1988.

Hwang, K. *Computer Arithmetic: Principles, Architecture and Design*. New York, NY: John Wiley, 1979.

*IBM System 370 Principles of Operation*. GA22-7000, IBM Corporation, 1973.

*IEE Computer Society. A Proposed Standard for Binary Floating-Point Arithmetic*. Draft 10, Task P754, December 1982.

Jouppi, N. P. and Sohi, G. S. *Readings in Computer Architecture*, San Francisco, CA: Morgan Kaufmann, 1999.
Lazou, C., *Supercomputers and Their Use*, New York, NY: Oxford University Press, 1988.
Levesque, J. M., Williamson, J. L., *A Guidebook to Fortran on Supercomputers*, San Diego, CA: Academic Press, Inc. 1989.
*MC6888/MC68882 Floating-Point Coprocessor User's Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1987.
Ortega, J. M., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, NY: 1988.
Patterson, D. A. and Hennessey, J. L. *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann, 1990.
Shiva, S. G. *Pipelined and Parallel Computer Architectures*, New York, NY: Harper Collins, 1996.
Stone, H. S. *High Performance Computer Architecture*, Reading, MA: Addison Wesley, 1993.
*The Bipolar Microcomputer Components Data Book*. Dallas, TX.: Texas Instruments Inc., 1978.
*The MSP430 Hardware Multiplier: Function and Applications*, Dallas, TX.: Texas Instruments Inc., 1999.
*The TTL Data Book*. Dallas, TX.: Texas Instruments Inc., 1986.

## PROBLEMS

10.1 List the architectural features of the ALUs of a mini-, micro-, and a large-scale system you have access to with reference to the characteristics described in this chapter.

10.2 Look up the TTL data book to find a carry look-ahead adder IC. Design a 16-bit carry look-ahead adder using an appropriate numer of these ICs.

10.3 Perform the multiplication of the following operands using the shift and add algorithm. Use the minimum number of bits in the representation of each number and the results.
   a. $24 \times 8$
   b. $42 \times 24$
   c. $17 \times 64$

10.4 Perform the following division operations using the restoring division algorithm. Use the minimum number of bits required.
   a. 24/8
   b. 43/15
   c. 129/23

10.5 Solve Problem 10.4 using nonrestoring division algorithm.

10.6 For the stack-based ALU shown in Fig. 10.5, derive the microoperation sequences needed for the following operations:

SHR: Shift the contents of top level right, once.
COMP: Replace the top level with its 2s complement. (Assume that the functional unit can perform the 2s complement of operand $X$.)
SUB: SL $\leftarrow$ SL $-$ TL, POP.

10.7 A hardware stack is used to evaluate arithmetic expressions. The expressions can contain both REAL and INTEGER values. The data representation contains a TAG bit with each operand. The TAG bit is 1 for REAL and 0 for INTEGER data. List the control signals needed for the stack-based ALU of Figure 10.5 to perform addition and subtraction of the operands in the top two levels of the stack. Include any additional hardware needed to accommodate this new data representation.

10.8 Give an algorithm to generate the sum of two numbers in excess-3 representation (i.e., each digit corresponds to 4 bits, in excess-3 format). Design a circuit for the adder similar to that in Fig. 10.7.

10.9 Develop the detailed block diagrams for each of the stages in the floating-point add pipeline of Fig. 10.8. Assume the IEEE standard representation for the floating-point numbers.

10.10 Develop the stages needed for a floating-point multiplier, assuming that the numbers are represented in the IEEE standard form.

10.11 A popular method of detecting the OVERFLOW and UNDERFLOW conditions in a shift-register-based stack is by using an additional shift register with number of bits equal to the number of levels in the stack and shifting a 1 through its bits. Design the complete circuit for an $N$-level stack.

10.12 Design the UNDERFLOW and OVERFLOW detection circuits for a RAM-based stack.

10.13 It is required to compute $C_i = A_i + B_i$ ($i = 1$ to 50), where $A$, $B$ and $C$ are arrays of floating-point numbers, using a six-stage add pipeline. Assume that each stage requires 15 ns. What is the total computation time?

10.14 Derive the formula for the total execution time of Problem 10.13 using a pipeline of $M$ stages, with each stage requiring $T$ ns. Assume that the arrays have $N$ elements each.

10.15 Describe the conditions under which an $n$-stage pipeline is $n$ times faster than a serial machine.

# 11
# Advanced Architectures

Up to this point we have considered processors that have essentially a single instruction stream (i.e., a single control unit bringing about a serial instruction execution) operating upon a single data stream (i.e., a single unit of data is being operated upon at any given time). We have described various enhancements to each of the subsystems (memory, ALU, control, and I/O) that increase the processing speed of the system. In addition, components from the fastest hardware technology available are used in the implementation of these subsystems to achieve higher speeds.

There are applications, especially in real-time processing, that make the machine throughput inadequate even with all these enhancements. It has thus been necessary to develop newer architectures with higher degrees of parallelism than the above enhancements provide in order to circumvent the limits of technology and achieve faster processing speeds.

Suppose that the application allows the development of processing algorithms with a degree of parallelism A, the language used to code the algorithm allows a degree of parallelism of L, the compilers retain a degree of parallelism of C, and the hardware structure of the machine has a degree of parallelism of H. Then, for the processing to be most efficient, the following relation must be satisfied:

$$H \geq C \geq L \geq A \tag{11.1}$$

Development of algorithms with a high degree of parallelism is application-dependent. A great deal of research has been devoted to developing languages that allow parallel processing constructs and compilers that either extract the parallelism from a sequential program or retain the parallelism of the source code through the compilation to produce parallel code. In this chapter, we will describe the popular hardware structures used in executing parallel code.

Several hardware structures have evolved over the years, providing for the execution of programs with various degrees of parallelism at various levels. We will describe a popular classification scheme for such architectures in the next section. Section 11.2 provides the details of selected parallel processing architectures. Section 11.3 provides a brief description of data-flow architectures, an experimental architecture type that provides the finest granularity of parallelism. Section 11.4 describes the concepts of distributed processing networks.

## 11.1 CLASSES OF ARCHITECTURES

Flynn divided computer architectures into four main classes, based on the number of instruction and data streams:

1. Single instruction stream, single data stream (SISD) machines, which are uniprocessors such as ASC, PDP-11, and INTEL 8080.
2. Single instruction stream, multiple data stream (SIMD) architectures, which are systems with multiple arithmetic-logic processors and a control processor. Each arithmetic-logic processor processes a data stream of its own, as directed by the single control processor. They are also called array processors or vector processors.
3. Multiple instruction stream, single data stream (MISD) machines, in which the single data stream is simultaneously acted upon by a multiple instruction stream. A system with a pipelined ALU can be considered an MISD, although that extends the definition of a data stream somewhat.
4. Multiple instruction stream, multiple data stream (MIMD) machines, which contain multiple processors, each executing its own instruction stream to process the data stream allocated to it. A computer system with one processor and an I/O channel working in parallel is the simplest example of an MIMD.

Almost all the machines used as examples in previous chapters are SISD machines; an exception is the CDC 6600, which is an MIMD with independent instruction streams, one for each active functional unit operating on multiple data streams. We will now provide models for the last three of the above-listed classifications followed by a brief description of several popular architectures.

### 11.1.1 SIMD

Figure 11.1 shows the structure of an SIMD. There are $n$ arithmetic-logic processors (P1 – P$n$), each with its own memory block (M1 – M$n$). The individual memory blocks combined constitute the system memory. The memory bus is used to transfer instructions and data to the control processor (CP). The control processor decodes instructions and sends control signals to processors P1 – P$n$. The processor interconnection network enables data exchange between the processors.

The control processor, in practice, is a full-fledged uniprocessor. It retrieves instructions from memory, sends arithmetic-logic instructions to processors, and executes control (branch, stop, etc.) instructions itself.

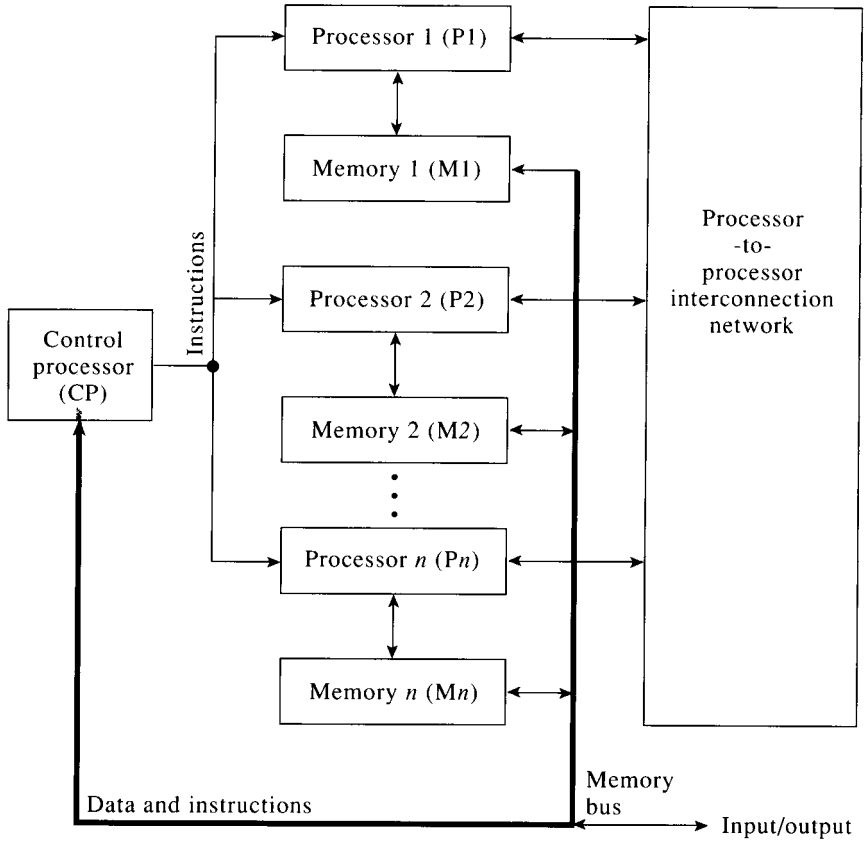



**Figure 11.1** SIMD structure

Processors P1 – P$n$ execute the same instruction at any time, each on its own data stream. Based on the arithmetic-logic conditions, some of the processors may be deactivated during certain operations. Such activation and deactivation of processors is handled by the control processor.

SIMDs are special purpose machines, suitable for array or vector processing. For example, in computing the sum of the column elements of a matrix, each column of the matrix can be assigned to a processor of an SIMD. The column of sums can then be computed in $N$ steps for an $N \times N$ matrix ($N \leq n$), rather than in $N^2$ steps required on an SISD. Thus, SIMDs can provide a high throughput, as long as the processing algorithm exhibits a high degree of parallelism at the instruction level.

### 11.1.2 MISD

Figure 11.2 shows the organization of an MISD machine. There are $n$ processors (or processing stations) arranged in a pipeline. The data stream from the memory enters the pipeline at processor 1 and moves from station to station through processor $n$, and the resulting data stream enters the memory. The control unit of the machine is shown to have $n$ subunits, one for each processor. Thus, there will be at any time $n$ independent instruction streams.

Note that the concept of data stream in this model is somewhat broad. The pipeline will be processing several independent data units at a given time. But, for the purposes of this model, all those data units put together is

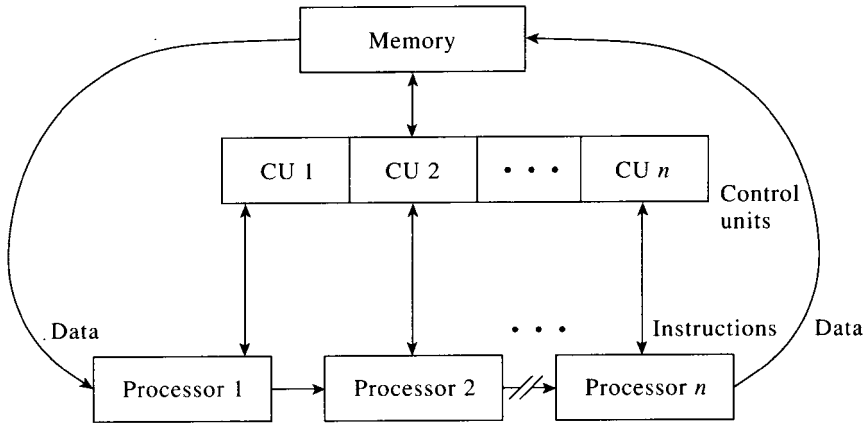



**Figure 11.2** MISD structure

considered a single data stream. As such, this classification is considered an anomaly. It is included here for the sake of completeness.

### 11.1.3 MIMD

Figure 11.3 shows an MIMD structure consisting of $p$ memory blocks, $n$ processing elements, and $m$ input/output channels. The processor-to-memory interconnection network enables the connection of a processor to any of the memory blocks. In addition to establishing the processor-memory connections, the network should also have a memory-mapping mechanism that performs a logical-to-physical address mapping. The processor-to-I/O interconnection network enables the connection of an I/O channel to any of the processors. The processor-to-processor interconnection network is more of an interrupt network than a data exchange network, since the majority of data exchanges can be performed through the memory-to-processor interconnection.

MIMDs offer the following advantages:

1. A high throughput can be achieved if the processing can be broken into parallel streams, thereby keeping all the processors active concurrently.
2. Since the processors and memory blocks are general-purpose resources, a faulty resource can be easily switched out, thereby achieving better fault tolerance.
3. A dynamic reconfiguration of resources is possible, to accommodate the processing loads.

MIMDs are more general-purpose in application than are SIMDs. The processors are not in synchronization instruction-by-instruction as in SIMD. But, it is required that the processing algorithms exhibit a high degree of parallelism, so that several processors are active concurrently at any time.

Some of the issues of concern in the design of an MIMD system are

1. Processor scheduling: efficient allocation of processors to processing needs in a dynamic fashion as the computation progresses.
2. Processor synchronization: prevention of processors trying to change a unit of data simultaneously; and obeying the precedence constraints in data manipulation.
3. Interconnection network design: the processor-to-memory or processor-to-peripheral interconnection network is still probably the most expensive element of the system and can become a bottleneck.

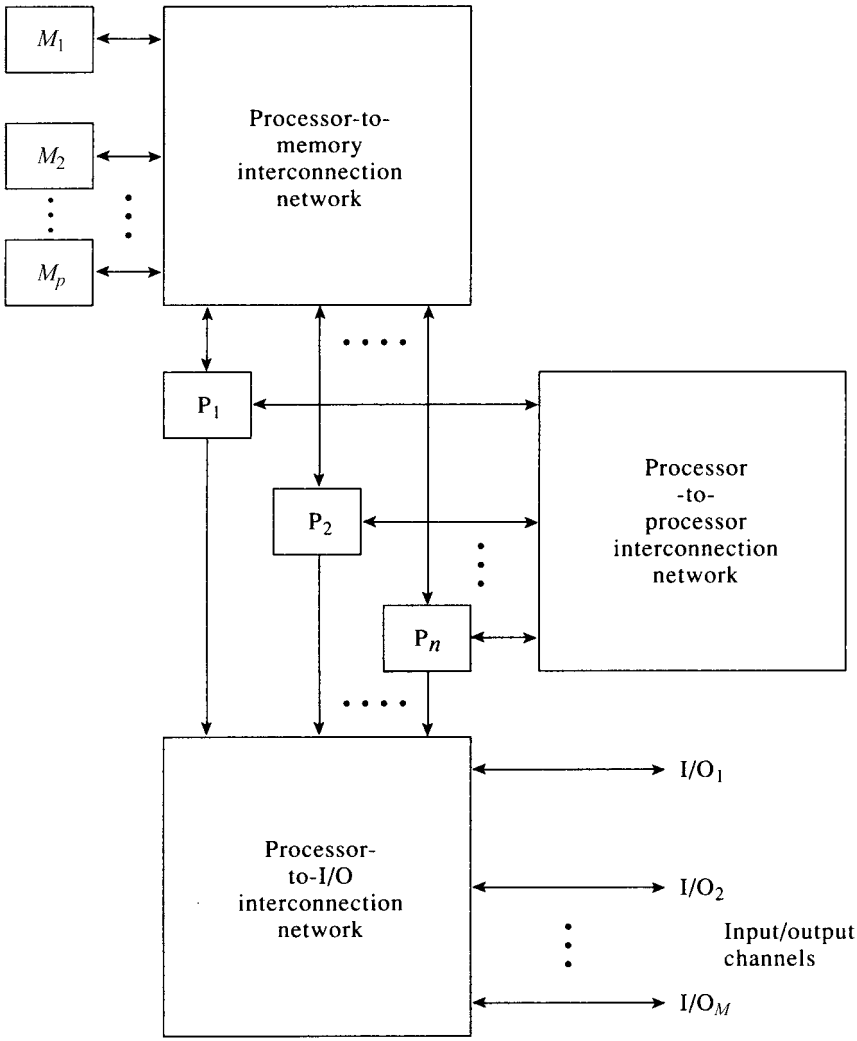



**Figure 11.3** MIMD structure

4. Overhead: ideally an $n$ processor system should provide $n$ times the throughput of a uniprocessor. This is not true in practice because of the overhead processing required to coordinate the activities between the various processors.
5. Partitioning: identifying parallelism in processing algorithms to invoke concurrent processing streams is not a trivial problem.

It is important to note that the architecture classification described in this section is not unique. A computer system may not clearly belong to one of these classes. For example, the CRAY series of supercomputers could be classified under any one of the four classes, depending on operating mode at a given time. Several other classification schemes and taxonomies have been proposed. Refer to *IEEE Computer* (November 1988) for a critique of taxonomies.

## 11.2 EXAMPLE SYSTEMS

Several of the example systems described earlier in this book operate in SIMD and MIMD modes to certain extent. The pipelining (MISD) structure is used extensively in all modern day machines. The Cray series of architectures operate in all the four modes of Flynn's classification, depending on the context of execution. In this section, we concentrate on examples of SIMD and MIMD architectures. We will call them *supercomputers*.

The traditional definition of supercomputers is that they are the most powerful computers available at any given time. This is a dynamic definition, in that today's supercomputers will not be considered supercomputers in a few years. All the machines designed for high-speed floating-point operations are usually classified as supercomputers. Based on their size, they can be further classified as high-end, mid-range and single-user systems. A machine that can perform 10 to 100 *million floating-point operations per second* (MFLOPS) is classified as a superminicomputer, while a machine with a rating of 100 to 2000 MFLOPS is a high-end supercomputer in today's technology.

Several supercomputer architectures have been attempted since the sixties to overcome the SISD throughput bottleneck. The majority of these architectures have been implemented in quantities of one or two. Nevertheless, these research and development efforts have contributed immensely to the area of computer architecture. We will briefly describe the following supercomputer architectures, concentrating mainly on architectural details rather than the application areas for these machines:

SIMD: ILLIAC-IV and Thinking Machines' Connection Machine.
MIMD: Cray T3E.

ILLIAC-IV is selected mainly for its historical interest. The Connection Machine series and the T3E represent the architectures of commercially available supercomputers.

### 11.2.1 ILLIAC-IV

The ILLIAC-IV project was started in 1966 at the University of Illinois. The objective was to build a parallel machine capable of executing $10^9$ instructions per second. To achieve this speed, a system with 256 processors controlled by a control processor was envisioned. The set of processors was divided into four quadrants of 64 processors each, each quadrant to be controlled by one control unit. Only one quadrant was built and it achieved a speed of $2 \times 10^8$ instructions per second.

Figure 11.4 shows the ILLIAC-IV system structure. The system is controlled by a Burroughs B-6500 processor. This machine compiles the ILLIAC-IV programs, schedules array programs, controls the array configurations, and manages the disk file transfers and peripherals. The disk file unit acts as the backup memory for the system.

Figure 11.5 shows the configuration of a quadrant. The control unit provides the control signals for all processing elements ($PE_0$–$PE_{63}$), which work in an instruction-by-instruction lock-step mode. The control unit (CU) executes all program control instructions and transfers processing instructions to PEs. The CU and the PE array execute in parallel. In addition, the CU generates and broadcasts the addresses of operands that are common to all PEs, receives status signals from PEs, from the internal I/O operations, and from the B-6500, and performs the appropriate control function.

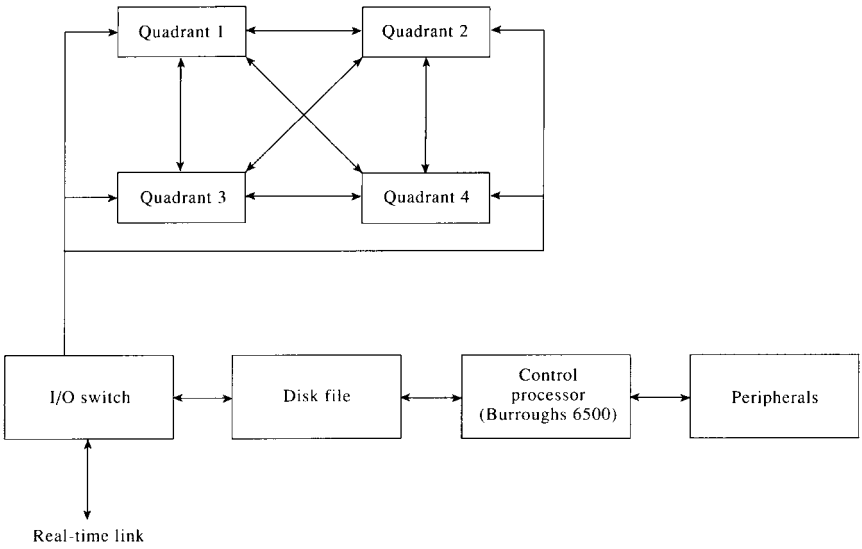
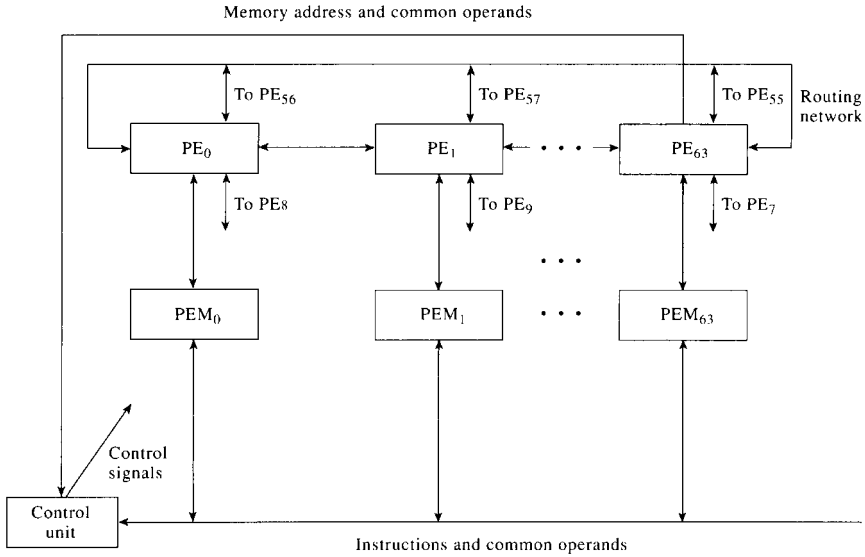



**Figure 11.4** ILLIAC-IV system structure




Note: $PE_i$ is connected to $PE_{i+1}$, $PE_{i+8}$, and $PE_{i-8}$;$0 \leq i \leq 63$.

**Figure 11.5** A quadrant of ILLIAC-IV

Each PE has four 64-bit registers (accumulator, operand register, data-routing register, and general storage register), an arithmetic-logic unit, a 16-bit local index register, and an 8-bit mode register that stores the processor status and provides the PE enable-disable information. Each processing element memory (PEM) block consists of a 250-nanosecond cycle-time memory with 2K 64-bit words.

The PE-to-PE routing network connects each PE to four of its neighbors (i.e., $PE_i$ to $PE_{i+1}$, $PE_{i-1}$, $PE_{i+8}$, and $PE_{i-8}$). The PE array is arranged as an $8 \times 8$ matrix with end-around connections. Interprocessor data communication of arbitrary distances is accomplished by a sequence of routings over the routing network.

The processing array is basically 64-bit computation oriented. But, it can be configured to perform as a 128 32-bit subprocessor array or a 512 8-bit subprocessor array. The subprocessor arrays are not completely independent because of the common index register in each PE and 64-bit data routing path.

The applications envisioned for ILLIAC-IV were

1. Manipulation of large matrices
2. Computations of solutions for large sets of difference equations for weather-prediction purposes

3. Fast data correlation for fast-response, phased-array radar systems.

Two high-level languages have evolved over the years for programming ILLIAC-IV: A FORTRAN-like language (CFD) oriented towards computational fluid dynamics and an ALGOL-like language, GLYNPNIR. A compiler that extracts parallelism from a regular FORTRAN program and converts it into a parallel FORTRAN (IVTRAN) has also been attempted. There is no operating system. ILLIAC-IV, being an experimental machine, did not offer a software library or a set of software tools, thereby making it difficult to program. The ILLIAC-IV is now located at the NASA Ames Center and is not operational.

### 11.2.2 Connection Machine-1

The Connection Machine is an SIMD with up to 65,536 processing elements. W. Daniel Hillis originated the concept for the Connection Machine in late 1970s while at Massachusetts Institute of Technology (MIT), with an aim to designing a machine that can efficiently simulate common-sense reasoning. The prototype machine was built in 1985, and it is currently manufactured by Thinking Machines Inc. The Connection Machine has been used in the following applications: memory-based reasoning, VLSI design, computer vision, and document retrieval. We will now provide a brief description of the Connection Machine architecture.

#### Major Hardware Components

The major components of a Connection Machine (CM) system are shown in Fig. 11.6. The configuration in the figure is only one of several possible combinations of components, but it does represent a system that might be used in a typical application. The core of the system is a parallel processing unit which may contain up to 65,536 processing elements. The processing elements execute instructions in parallel and can exchange data with one another through hardware interconnections using a packet-switched message routing mechanism. Data and instructions for the parallel processing unit arrive from a crosspoint switching network, called the nexus, after being generated in a front-end processor. Up to four front-end processors, each a DEC VAX or Symbolics 3600 series computer, may be used in a single CM system. Because identical instructions are executed simultaneously on many processing elements, the CM is classified as a single-instruction multiple-data (SIMD) machine. The I/O subsystem components

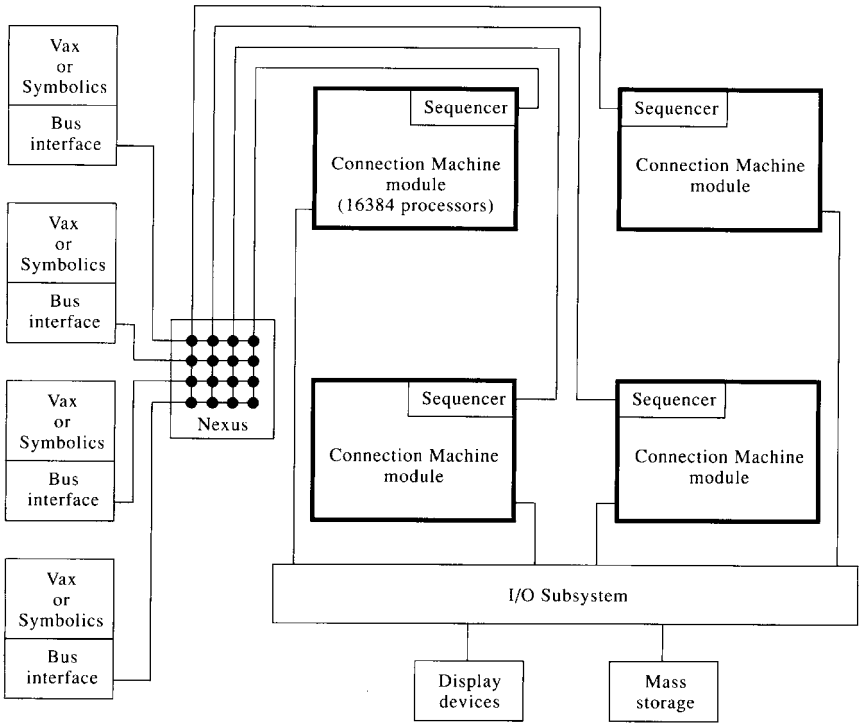



**Figure 11.6** Connection Machine hardware components (Courtesy of Thinking Machines Inc.)

provide a high-speed interface for mass storage and visual output devices. The application of parallelism is as apparent in the I/O subsystem design as it is in the design of the parallel processing unit. Very high data rates are achieved by using parallel disk units – one for each bit in a 32-bit I/O data path. Error detection and correction is provided that adds seven bits to the 32-bit data path and seven disk units to the mass storage equipment.

Most of the following discussion on CM architecture will concentrate on the properties of the parallel processing unit and will include only brief descriptions of the I/O subsystem, front end processors, and any other units that are exteral to it. The hardware descriptions apply specifically to a model CM-1 Connection Machine, the original commercial model described by Hillis (Hillis, 1985) and by Tucker (Tucker, 1988). Tucker also discusses the CM-2, an improved and partially redesigned machine with increased processing speed and memory capacity. Although the CM-2 differs from

the CM-1 in some of the ways functions are implemented, the principles of operation are essentially the same as in the CM-1.

The latest in the series of connection machines is CM-5. This is a hybrid SIMD/MIMD architecture. We now provide a brief description of CM-1.

### Hardware Organization and Physical Characteristics

The parallel processing unit is configured with one to four modules, each containing 16,384 processing elements and a sequencer. The sequencer is a microprogrammed controller that interprets microinstructions from the front-end processors and converts them to nanoinstructions for use by the processing elements. Within a module, 16 circuit boards each hold 32 custom chips and associated memory and logic devices. Each custom chip contains 16 processing elements, a control unit, and a router unit (Fig. 11.7). A complete circuit board module is assembled and housed in the shape of a cube approximately 1.5 meters on a side. A light-emitting diode for each of the 16K processing elements is installed on a side of the cube to assist in troubleshooting and monitoring of processor operation (Hillis, 1987).

### Processing Element Operation

A block diagram of one of the sixteen processing elements on a CM custom chip appears in Fig. 11.8. The memory associated with the processing element is also shown, although it is physically located on a separate chip. The bit-addressable memory supplies two bits to the input of the ALU. A third bit for the ALU input is taken from one of sixteen one-bit flag registers. The ALU generates two output bits, one of which can be stored in memory while the other can be stored in one of the flag registers. Signals from the control unit located on each CM module determine:

1. Whether an instruction will be executed or not
2. What memory locations and which one of the flags are used for ALU input
3. What memory location and flag receive the ALU output
4. Which operation the ALU executes.

Whether or not an instruction is executed depends on the state of a condition flag. Four bits from the control unit specify which of the processor element flags will be used as a condition flag, and one bit indicates whether a 1 or a 0 is the true state of the flag. This provides a powerful, general

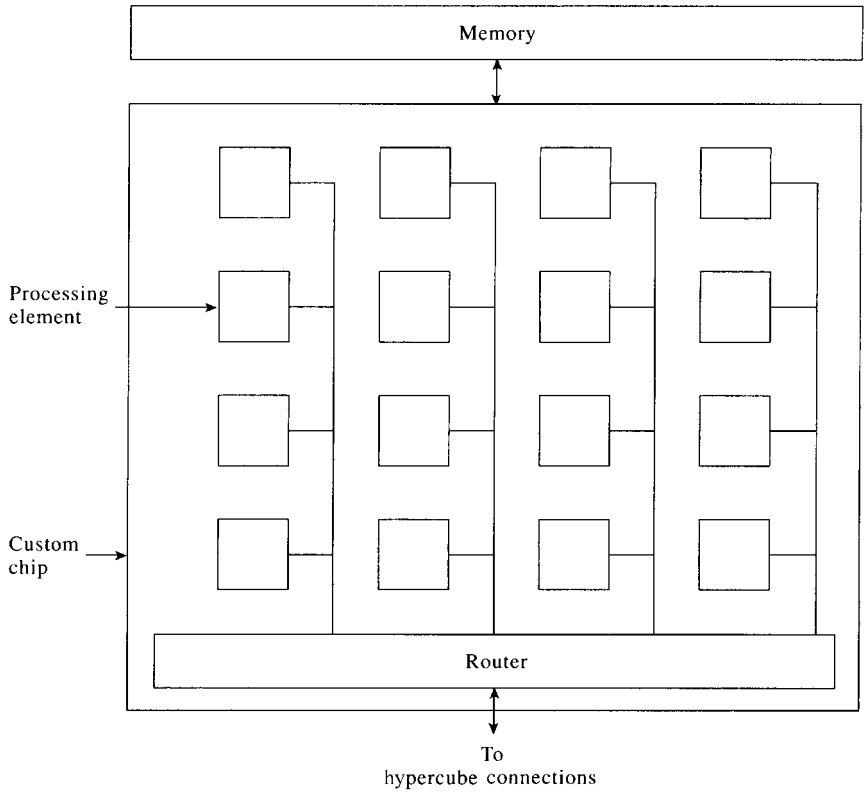



**Figure 11.7** Connection Machine custom chip (Courtesy of Thinking Machines Inc.)

mechanism for executing each instruction based on the condition of any of the sixteen flags in either the 1 or 0 state.

ALU operations are specified by a 16-bit control path. Eight bits of the 16 are called the memory truth table and the other eight form the flag truth table. These two truth tables select one of 256 possible boolean operations to apply to the memory and flag ALU input bits respectively. Two 12-bit address paths from the control unit select the two ALU memory bit inputs and two 4-bit paths select the ALU flag bit input source and output destination. Two more bits from the control unit specify a north, east, west, or south (NEWS) direction for data to move to or from an adjacent processing element during each instruction. The NEWS network functions (Fig. 11.9) allow movement of data one bit at a time to adjacent processing elements,

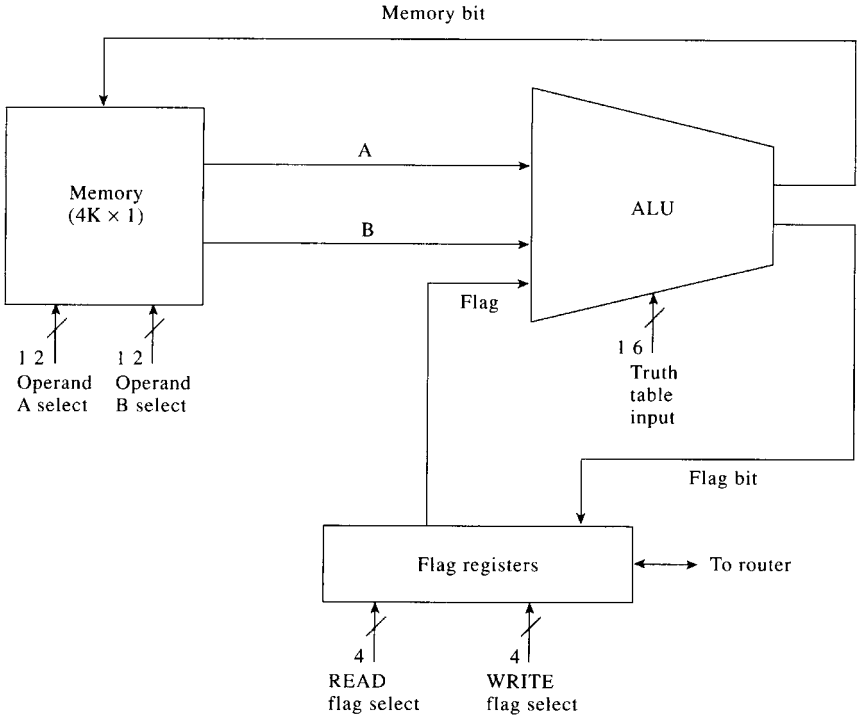



**Figure 11.8** Connection Machine processing element (Model CM-1). (Courtesy of Thinking Machines Inc.)

all of which are connected on-chip. Each chip can be connected to other chips to form a two-dimensional array.

The 16 flag bit registers include eight bits for general purpose use and eight bits that have functions predefined by the hardware. Special purpose flags include the NEWS flag (which can be written to/from the ALU flag output of one of the four adjacent processing elements), two flags for message router data movement and handshaking, a memory parity flag, a flag for daisy chaining processing elements, a zero flag for reading or for operations that do not need to write to a flag, and two flags used primarily for diagnostics.

## Data Movement

The ability to rapidly move data from one processing element to another is essential for efficient operation of the CM. Several mechanisms are provided

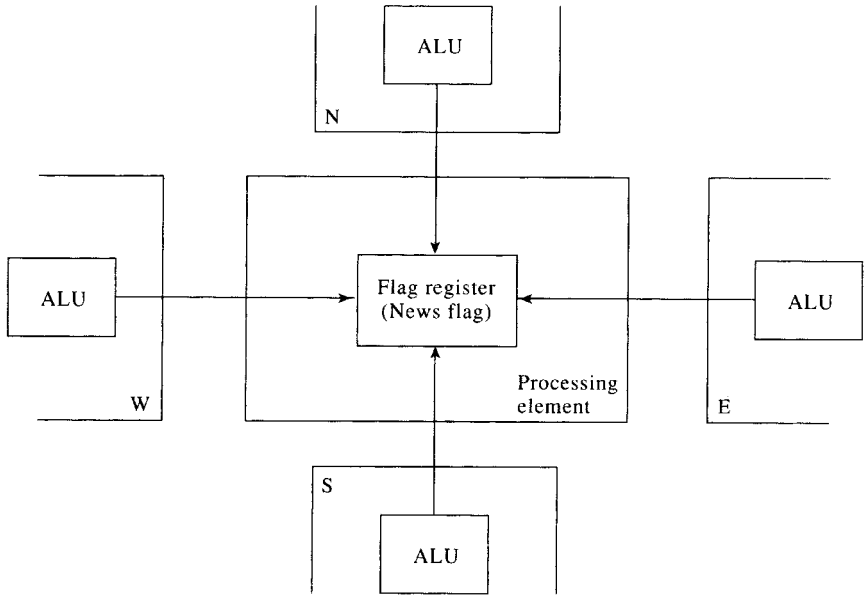



**Figure 11.9** News network (Courtesy of Thinking Machines Inc.)

in the CM for moving data. For a given computational problem one mechanism may be more suitable than another: the most efficient method can be selected by an algorithm designer to make the best use of the CM for each application.

The NEWS network previously discussed provides a very fast way to move data to neighboring processing elements. Problems with simple, repetitive data patterns may benefit from the use of the network.

A slower but much more flexible communication mechanism is provided by the message router. The message router uses the router controller in each custom processing element chip as a node in a packet-switching network. In a four-module CM with 64K processing elements, 4096 router controllers are connected in a 12-dimensional boolean $n$-cube, or hypercube. This configuration is such that no more than 12 paths (edges of the hypercube) must be traversed for the message packet from one router controller to reach any other router controller. Message packets are serial data streams that contain a destination address field and a data field. Contents of the data field are loaded into the memory of the destination processing element. This method of communication allows complex bit patterns to be moved between any two processing elements. Hillis provides a comprehensive description of

the message router network as well as an analysis of performance of the network under various operating conditions.

The hypercube connection scheme is also used to implement parallel operations under control of the sequencers. A sort algorithm using hypercube communication has been used to sort 64K 32-bit keys in approximately 30 ms (Tucker, 1988).

Tucker also mentions briefly that another communication mechanism used in the CM is a broadcast facility that allows data to be transmitted to all processing elements at once from the front-end processors or from the sequencers.

One interesting aspect of processing element operations is that instructions are executed at speeds that are much slower than speeds commonly encountered in the CPU hardware of conventional computers. The clock speed used to synchronize instruction execution is 4 MHz, and a typical instruction cycle of fetch, execute, store takes 750 ns. Compensation for the slow execution of processing element instructions is provided by the massive parallelism which effectively multiplies the total number of instructions executed by a very large factor.

### Programming the Parallel Processing Unit

Because of the parallel construction of the CM, the techniques for low-level programming of the processing unit are considerably different from the techniques suitable for programming computers with conventional architectures. Programming the CM at the assembly level is done with a parallel instruction set language called Paris. Paris provides the programmer with instructions that make use of the parallel nature of the processing unit, the processing element communications networks, and the I/O subsystem. The sequencers in each CM module are programmed with Paris, and high-level language compilers running on the front-end processors generate code files containing Paris instructions. One of the most important features implemented by Paris is the virtual processor mechanism. This feature allows the programmer or user to specify the use of more processing elements than are actually available in the CM. For example, if a million processors are needed for a problem, Paris provides a mechanism, which is transparent to the user, for solving the problem as if the one million processors were actually available.

Instructions are also available in Paris for operations that make direct use of the processing element communication networks. The sort algorithm (discussed earlier) is implemented with hypercube communications and makes use of instructions of this type.

### Performance

The CM architecture provides massive parallelism, a very general instruction mechanism, and efficient data communications between processing elements. However, the CM must be evaluated for its performance while solving practical problems in order to determine the "success" of the architecture in comparison.

Hillis provides the following indicators of performance for the CM-1 models:

- Memory size: $2.5 \times 10^8$ bits
- Memory bandwidth: $2.0 \times 10^{11}$ bits/s
- Processor bandwidth: $3.3 \times 10^{11}$ bits/s
- I/O bandwidth: $5.0 \times 10^8$ bits/s
- Arithmetic operations on 32-bit integers: 1000 MIPS (million instructions per second)

The newer model CM-2 provides a significant increase in performance when compared to the CM-1. The 32-bit integer arithmetic and logical operations can run at 2500 MIPS. The CM-2 also contains floating-point hardware not used in the CM-1. This hardware helps the CM-2 exceed 20 billion floating-point operations per second (20 GFLOPS) on problems not requiring interprocessor communications and 5 GFLOPS on problems that must use the processing element communications networks. An example of the latter type of problem is the multiplication of two 4000-by-4000-element matrices. I/O bandwidth of the CM-2 allows a sustained rate of 210 MB/s between the parallel processing unit and disk storage.

### Software

Software development for the Connection Machine remains primarily in the research and academic worlds. Software for the complex parallel architectures is far from being in reach of the general programming public. Languages currently used on the Connection Machine are C*, *LISP, CM LISP and FORTRAN 8x. These are conservative variations of the C and LISP languages that contain additional sets of instructions and parallel structures.

### LISP

An example of a parallel instruction in *LISP is: (+!!*ab*) (Tucker, 1988). This instruction forms the addition of the variables *a* and *b* on each of the 64K processors and stores the result on a stack within each processor in

approximately 300 $\mu$s. For problems that are best divided into more than 64K units, the Connection Machine uses its virtual processor mechanism to allow the programmer to process data structures with many more than 64K elements. For example, a problem requiring the addition of two vectors with a million elements each could be solved with a program that assumes the availability of 1M processors.

Another dialect of Common Lisp, Connection Machine Lisp, was written to support the fine-grained data-oriented style of the Connection Machine but can also be implemented on other parallel and sequential computers (Steele and Hillis, 1986).

In Connection Machine Lisp, all parallel functions are based on the xapping (rhymes with mapping) data structure, which is similar to a sparse matrix except that its elements can be processed in parallel. A xapping is an unordered set of ordered pairs. A pair consists of an index and a value written as index → value. Both the index and value may be any Lisp object. This is an example of a xapping:

{apple → red;sky → blue;grass → green;orange → orange}

The index corresponds to a label for a processor with the value being stored in the processor's memory. If the indices happen to be a finite set of consecutive integers beginning with 0, then the notation may be abbreviated so that only the values must be given in square brackets. Then the xapping is called a xector. In this case,

{0→ red;1 → green;2 → yellow} = = [red green yellow].

The $\alpha$ notation is used to apply execution of a given function to all elements of the sequence. For example, ($\alpha$sqrt '[1 4 9 16]) = = [1 2 3 4]. An infinite xapping can be produced by ($\alpha$ 7). This will cause every processor to be assigned the constant 7. When the operation has completed on the slowest processor, the processors are resynchronized.

Conditions such as "($\alpha$ if p $x$ $y$)" may also be executed in parallel. For indices in which p has true values, $x$ will be evaluated, and for indices in which $p$ has a false value, $y$ will be evaluated.

The language continues to be developed with remaining conflicts between several design goals such as compatibility with Common Lisp vs. convenience in using functional arguments. The control over parallelism and the generality of the $\alpha$ notation remain to be controversial design issues.

### Programming Philosophy

Programming the Connection Machine is easy due to the fact that Symbolics and DEC-Vax machines are used as front-end interfaces to allow the user to work with familiar operating systems, networks, and file systems. In addition, the compilers are written so that sequential code is converted into parallel code. However, as illustrated above, software developers must take a different approach if they are to utilize the full potential of this exotic parallel machine.

Developing new software in a parallel version of a familiar language probably will not cause culture shock, but a major problem would be to rehost existing code onto a Connection Machine. Programs for existing applications have sequential methods deeply embedded in their logic which makes the conversion of existing code difficult. To do so, it would be necessary to divide the code by hand into blocks to be assigned to each processor. However, the Connection Machine requires the programmer to analyze existing code to determine data flow and dependencies and to arrange the code in such a way as to achieve the optimum speed possible.

Rather than trying to revamp these programs, many will have to be rewritten completely using the data-parallelism method to exploit the performance advantages of the new machine. It is believed, however, that the price advantages will be great enough to pay for this undertaking.

New programming techniques must be learned, the nature and extent of which are controversial. The main challenge will be to create and utilize the algorithms that most efficiently control the parallel operation of processors.

## 11.2.3 Thinking Machine Corporation Connection Machine 5 (CM-5)

The CM-5 is a hybrid MIMD/SIMD multiprocessor system. The number of processors can be scaled from 32 to 16,384 providing an observed peak performance of 20 GFLOPS. Optionally, four vector units can be added to each processor, resulting in a peak performance of 2 TFLOPS. The CM-5 is controlled by one or more front-end workstations (Sun Microsystems' SPARC 2, Sun-4, or Sun-600), that execute the CM-5 operating system, CMost. These front-end workstations (control processors) are dedicated to either PE array partition management or I/O control, and control all the instructions executed by the CM-5.

Figure 11.10 shows the system block diagram of the CM-5. There are three independent network subsystems – the Data Network, the Control Network, and the Diagnostic Network. All system resources are connected

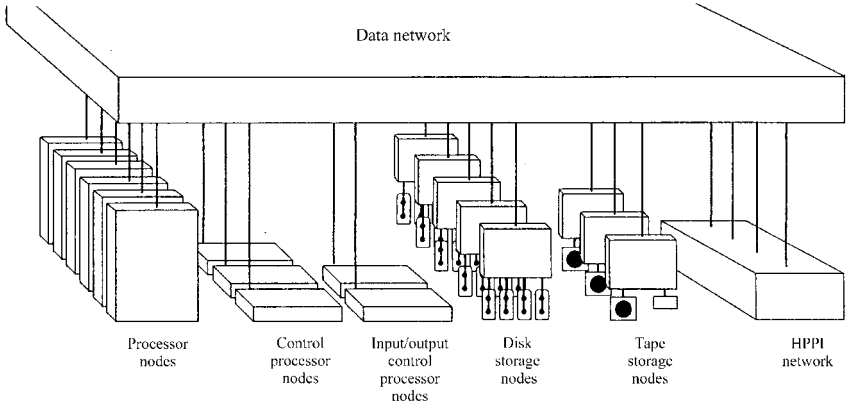



**Figure 11.10** CM-5 system block diagram (Reprinted by permission of Thinking Machines Corporation.)

to all these networks. Processing nodes are not interrupted by network activity, which permits message routing and computational tasks to be performed in parallel. The control processor broadcasts a single program to all the nodes. Each node runs this program at its own rate, utilizing the control network for synchronization. The library primitives provided with the machine, allow the implementation of synchronous communications for data parallel parts and asynchronous communications for the message-passing parts of the program.

The Data Network provides for data communications between CM-5 components. It has a fat tree topology. Each set of four processing or storage nodes has a network switch connecting them together. This switch provides for the data communication between the four nodes and to the upper layers of the network. Four of these groups of processing or storage nodes are in turn connected by a higher-level switch. Four of these larger groups are in turn similarly connected, to provide an aggregate bandwidth of 1280 MB/s. At the bottom layer, each node is connected to its corresponding network switch by two bidirectional links of 5 MB/s capacity.

The control network subsystem coordinates the processor interactions required to implement inter-processor broadcasting and synchronization. It also provides the interface between the CM-5 front end processors and the corresponding processor nodes, and provides protection against multi-user interference.

The diagnostic network subsystem provides the interfaces required to perform internal system testing of device connectivity and circuit integrity.

It can also access machine environmental variables (temperature, air flow, etc.) for detection of failure conditions.

Each processing node consists of a 22 MIPS SPARC microprocessor, 32 MB of memory, four 32 MFLOP vector processing units (64-bit operands) each with 8 MB memory, and a network interface (Fig. 11.11). The vector units implement a full floating-point instruction set with fast divide and square root functions. The SPARC is the control and scalar processing resource of the vector units. It also implements the operating system environment and manages communication with other system components via the network interface.

The scalable disk array is an integrated and expandable disk system that provides a range of 9 GB to 3.2 TB of storage. Data transfers between CM-5 and the Disk Array can be sustained at 12 MB/s to 4.2 GB/s. The disk array is an array of disk storage nodes. Each node contains a network interface implemented by a RISC microprocessor controller, disk buffer, four SCSI controllers, and eight disk drives. The scalable disk array features are implemented via the vendor's CM-5 scalable file system (SFS) software, which executes on a SPARC-based I/O control processor.

The I/O nodes are the third class of computational resources in a CM-5 system. These nodes include magnetic tape and network communications facilities. The integrated tape subsystem is implemented via a specialized I/O node which has local buffer memory and a pair of SCSI-2 channel controllers that can connect to a maximum of seven devices per controller. Communication with other computers is provided by the HIPPI I/O node. Serial and parallel TCP/IP and user HIPPI-FP style interfacing through sockets is provided by the HIPPI subsystem. CM-5 also supports Ethernet and FDDI communications.

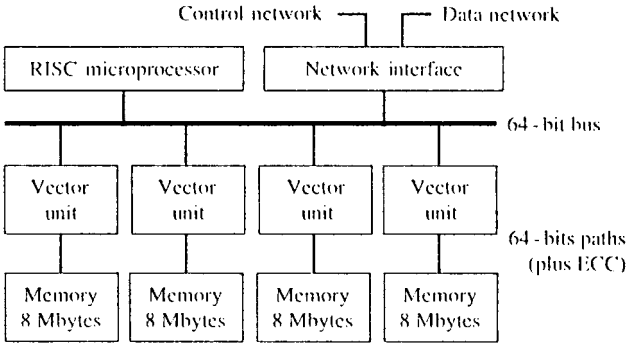



**Figure 11.11** Block diagram of a processing node (Reprinted by permission of Thinking Machines Corporation.)

CMost provides the integrated user interface in the form of the X-Window System and OSF/Motif. The software environment integrates the debugging, performance analysis, and data visualization facilities into the integrated user interface. The following languages are available: C*, CM FORTRAN, and *Lisp, which are respective supersets of C, FORTRAN-90, and Lisp.

### 11.2.4 Cray T3E System

The Cray T3E system is a scalable shared-memory massively parallel processor (MPP) which is a follow-on to the Cray T3D series of systems. The processor used in this system is Compaq's Alpha 21164 (originally designed by Digital Equipment Corporation) which is a RISC architecture.

The interconnection network used in this system is a bidirectional 3D torus (it retains this from the T3D system). It is aimed at minimizing latencies and maximizing bandwidths between processing nodes and supports scalability up to 2048 nodes (processing elements).

The system performs I/O through multiple ports onto one or more scalable GigaRing channels, for intersystem and system-to-peripheral communication with maximum transfer bandwidth of around 500 MB/s.

The T3E is a distributed shared memory system. The advantage of this type of system over traditional shared memory vector systems or message-passing systems is that a large number of processors may be involved in solving a problem at any given efficiency, and a finer granularity of work can be performed with a given number of processors. This is because distributed shared memory allows the main task to be partitioned into fine-grained subtasks without greatly increasing the explicit (programmer-introduced) synchronization requirements and communication overhead.

The T3E system can be either water-cooled or air-cooled. It is currently offered in configurations of from 6 to 128 processing elements for air-cooled systems and 32 to 2048 processing elements for liquid-cooled systems.

With peak performances of 120 MFLOPS per processor on the latest T3E-1200 systems, a fully loaded Cray T3E can attain a total peak performance of 2.5 TFLOPS (teraflops). The Cray T3E-1200 system is priced starting at $630 000 (for an air-cooled system with six processing elements).

The Cray T3E series is extensively used in solving computationally intensive problems in the scientific and technical areas for a wide range of applications including electromagnetic, chemistry, fluid dynamics, weather simulation/prediction, 3D oil exploration, and 3D seismic processing.

## Hardware

A *node* in a Cray T3E system (also called a *processing element*, or *PE*) consists of the following components:

- Alpha 21164 microprocessor
- System support circuitry (primarily the system control chip)
- Local memory (64 MB to 2 GB)
- Network router.

Figure 11.12 shows the components of a node in the T3E and the connections among these components.

Each processing element may be enabled or disabled individually. Extra nodes can be incorporated into the system, and can be used to replace a faulty node using a hot-swap mechanism. This can be done without rebooting the system.

One *motherboard* consists of four such nodes and one input/output controller. The network routers in the system are connected in a 3D torus interconnect; the input/output controllers are connected to the interconnection network by connections to each of the four local network routers on the motherboard and to external devices by connections to a high speed GigaRing channel. In the basic configuration, every two input/output controllers need to be attached to a GigaRing channel.

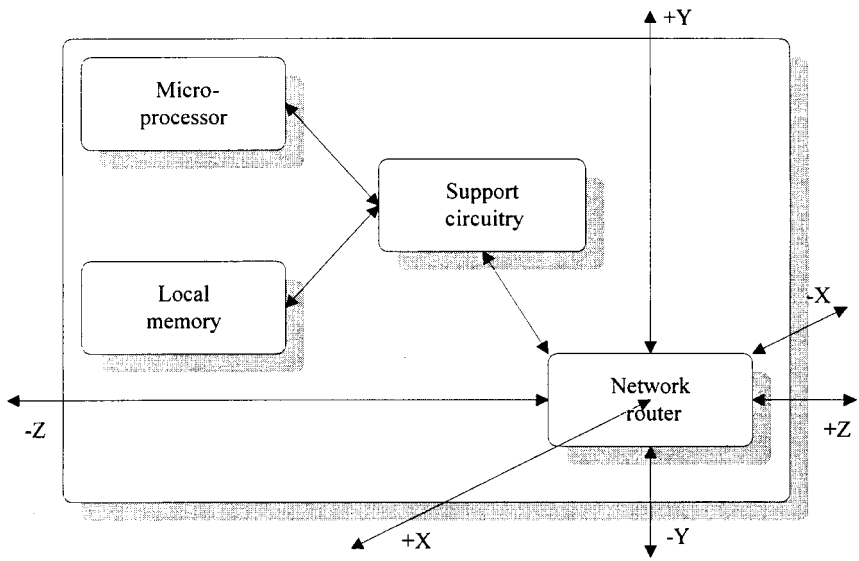



**Figure 11.12** Components of a T3E node (Haataja and Savolainen, 1998)

The following subsections describe each component of the T3E node.

The microprocessor in each node of the T3E, as stated earlier, is an Alpha 21164. This is a cache-based, superscalar four-way instruction issue, 64-bit load and store RISC processor with pipelined functional units. The registers are 64 bits long and all operations are performed between these registers. The instructions are all 32 bits long and all memory operations are either load or store operations. It supports the following data types:

- 8-, 16-, 32-, and 64-bit integers
- IEEE 32-bit and 64-bit floating-point formats
- VAX architecture 32-bit and 64-bit floating-point formats.

The 21164 supports a 40-bit physical address size and a 43-bit virtual address size. Virtual addresses seen by the program are translated to the physical address by the memory-management mechanism. The data bus is 128 bits wide. The microprocessor implements a cacheable memory space and a non-cacheable input/output space and it uses the most significant bit of the physical address (the 39th bit) to distinguish between these spaces. Local memory loads and stores use the cacheable memory space. Figure 11.13 shows the architecture of the Alpha 21164, and Table 11.1 lists the specifications of an Alpha 300 MHz system. This section will discuss the instruction fetch/decode and branch unit, memory address translation unit, floating-point execution unit, integer execution unit, and the cache control and bus interface unit. The three cache units (instruction cache, data cache, and secondary cache) are discussed in Section 0 below.

The main function of the *instruction fetch/decode and branch unit (IDU)* is to manage and issue instructions to the integer execution unit, the floating-point unit and the memory address translation unit. The IDU also manages the instruction cache. It primarily comprises:

- Instruction pre-fetcher, decoder and buffer
- 48-entry instruction translation buffer
- Program counter and branch prediction logic
- Instruction slot and issue logic
- Interrupt, exception, and abort logic.

The IDU can decode up four instructions in parallel. The IDU issues multiple instructions in parallel only if the resources required by all the instructions are available. The IDU only issues instructions in order; hence if resources are available for a later instruction but not for an earlier one, the later instruction will not be issued until the earlier instruction has been issued. The IDU does not advance to a new group of four instructions until all four instructions in the current group have been issued.

**Table 11.1** 21164 Microprocessor Specifications

| Features | Description |
|---|---|
| Processor | Alpha 21164 RISC |
| Transistor count | 9.67 million |
| Physical address size | 40 bits |
| Virtual address size | 43 bits |
| Bus | Separate data and address bus, 128 bit data bus |
| Clock rate | 300 MHz |
| Page size | 8 KB |
| Issue rate | Two integer instructions and two floating-point instructions per cycle |
| Integer instruction pipeline | Seven stages |
| Floating-pt instruction pipeline | Nine stages |
| Peak performance rate | 1200 MIPS |
| Peak floating-point performance | 600 MFLOPS |
| On-chip L1 Dcache | 8 KB, physical, direct-mapped, write-through, 32-byte block, 32-byte fill |
| On-chip L1 Icache | 8 KB, virtual, direct-mapped 32-byte block, 32-byte fill, 128 address space numbers |
| On-chip L2 Scache | 96 KB, physical, 3-way set associative, write-back, 64-byte block, 64-byte fill |
| On-chip data translation buffer | 64-entry, fully associative, not-last-used replacement, 8K pages |
| On-chip instruction translation buffer | 48-entry, fully associative, not last-used replacement |
| Latency of a Dcache hit | Two clock periods |
| Latency of a Scache hit | 8–10 lock periods |

The IDU contains an instruction pre-fetcher and a four-entry (32 bytes per entry) pre-fetch buffer called the *refill buffer*. A miss in the instruction cache is first looked for in the refill buffer; if present, it is loaded into the Icache immediately. If the refill buffer does not contain the necessary information, then the appropriate fetch request and a number of pre-fetches are sent to the memory address translation unit. One pre-fetch is sent per cycle until all four entries are occupied, taking into account any previously sent and pending fills.

A two-bit *history state* is used to record the outcome of branch instructions for each instruction in the instruction cache. It is a saturating counter that gets incremented when a branch is taken and decremented when it isn't.

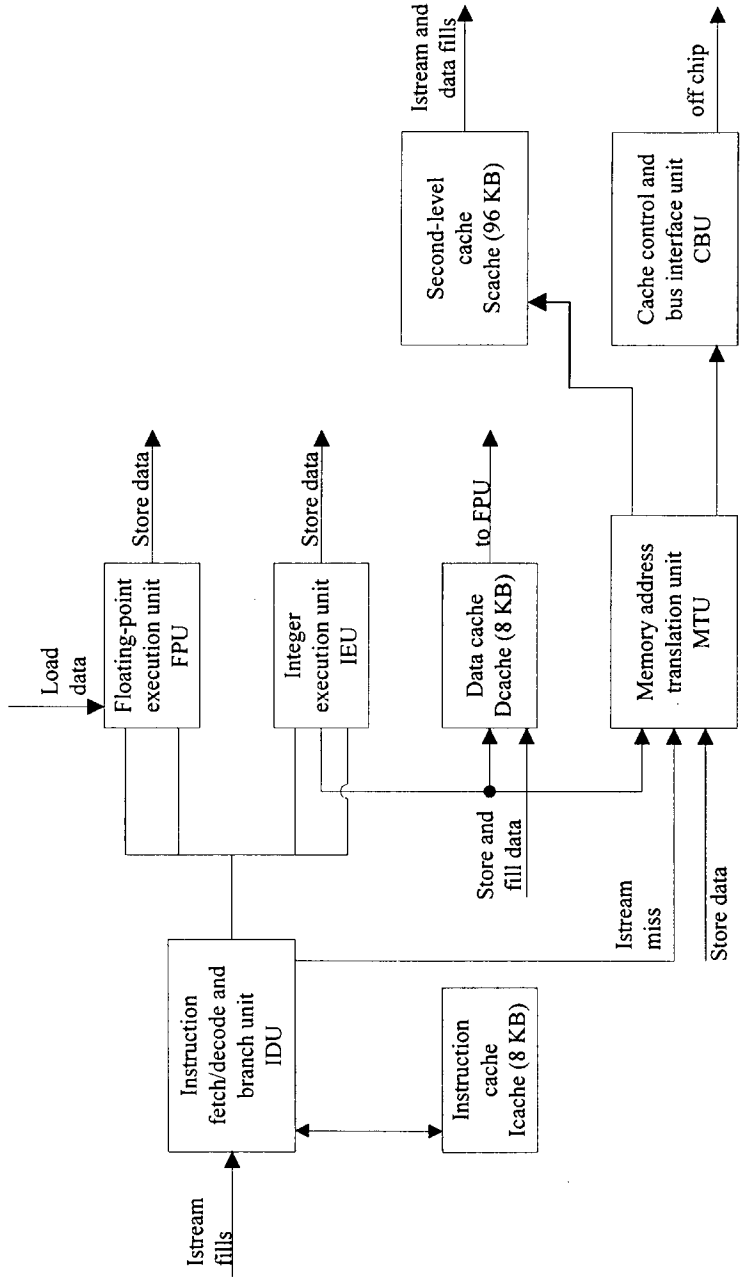



**Figure 11.13** Alpha 21164 microprocesor (Anderson et al., 1997)

The history state is used to predict the outcome of the next execution of a branch instruction.

The IDU also contains an *instruction translation buffer* which has 48 fully associative entries. This buffer stores the most recently used Instruction stream address translations.

The *floating-point execution unit (FPU)* consists of a floating-point add pipeline, a floating-point multiply pipeline, a user-accessible control register and a 32-entry, 64-bit floating-point register file. The floating-point divide unit is associated with the add pipeline, but is not itself pipelined. Except for floating-point divide instructions, the FPU can accept two instructions per cycle. Both the add and multiply pipelines have nine stages. The floating-point register file has five read ports and four write ports. Four of the read ports are used by the two pipelines to load operands, while the fifth is used for floating-point stores. Two of the write ports are used to write results from the two pipelines while the remaining two ports are used for writing fills from floating-point loads.

The *integer execution unit (IEU)* contains two 64-bit, seven-stage integer pipelines (E0 and E1) that contain two adders, two logic boxes, a barrel shifter, byte-manipulation logic, an integer multiplier, and a 40-entry 64-bit integer register file. This register file has four read ports and two write ports. The read ports are used to provide operands to both the pipelines while the write ports are used to store results from the pipelines.

The *memory address translation unit (MTU)* is made up of three main components: the data translation buffer, memory address file, and write buffer. It controls the data cache.

The *data translation buffer* is a 64-entry fully associative buffer that stores the most recently used data-stream page table entries and uses a not-last-used replacement algorithm.

The *memory address file* provides buffering capabilities for load requests that miss in the data cache before being sent to the second-level cache. It has six 32-byte entries for load misses and four 32-byte entries for IDU instruction fetches and pre-fetches. When a data cache miss occurs, this miss is compared with each entry in the memory address file. If any entry in the file contains a load miss that addresses the same address as the current miss and certain merging rules are satisfied, then the two load requests are merged into one, implying that two or more data cache misses can be serviced with one data fill.

The *write buffer* provides buffering capabilities for store requests that miss in the data cache before being written to the secondary cache. Note that just like the memory address file, each entry of the write buffer can satisfy requests of one or more store instructions (i.e., merging is allowed). The write buffer has six 32-byte entries.

All of these buffering capabilities are required since the processor can issue load/store requests at a rate faster than the rate at which the secondary cache can supply them, particularly when the secondary cache misses.

The *cache control and bus interface unit (CBU)* processes all requests from the memory address translation unit and performs all memory-related functions, especially enforcing the coherence protocols for the write-back caching mechanism. It controls the secondary cache. It forms an interface between the processor and the support circuitry. It also controls the 128-bit bidirectional data bus, address bus, and input/output control.

The 21164 can issue up to a maximum of four instructions per clock cycle, giving a peak performance of 1200 MIPS and 600 MFLOPS on the original Cray T3E machine. Table 11.2 gives the peak performance per processor of the three different T3E systems.

### Local Memory

The memory hierarchy in a T3E node is composed of the following levels:

- Registers
- On-chip level one (L1) cache
- On-chip second-level (L2) cache
- Main memory (DRAM).

Figure 11.14 shows the local memory hierarchy of a T3E node.

The level one cache consists of an 8 KB direct mapped data cache (Dcache) and an 8 KB direct mapped instruction cache (Icache). Both caches are direct-mapped, and are split up into 256 lines, making each line 32 bytes long.

Second-level cache (Scache) is a three-way associative 96 KB memory that caches both instructions and data. It contains three sets of 512 64-byte lines.

Main memory (DRAM) is divided up into eight memory banks, and each line in a bank is a 64-bit word. There are four memory controllers that

**Table 11.2** T3E Performances Per Processor

| Series | Processor speed (MHz) | MFLOPS | MIPS |
|---|---|---|---|
| Cray T3E | 300 | 600 | 1200 |
| Cray T3E-900 | 450 | 900 | 1800 |
| cray T3E-1200 | 600 | 1200 | 2400 |

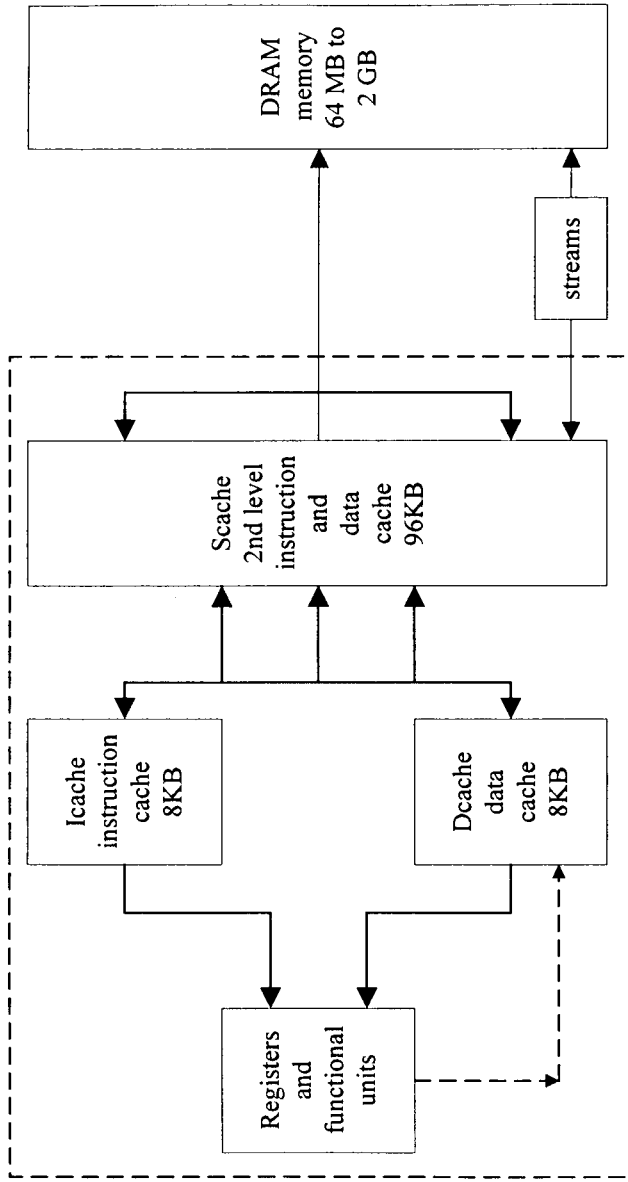

DRAM memory 64 MB to 2 GB
streams
Scache 2nd level instruction and data cache 96KB
Icache instruction cache 8KB
Dcache data cache 8KB
Registers and functional units

**Figure 11.14** Local memory hierarchy (Haataja and Savolainen, 1998)

control these banks. This means that one 64-byte Scache line is spread across eight banks, each bank containing one 8-byte portion. The memory in a single processor can vary from 64 MB up to 2 GB.

In the event of a data hit, the latency of data transfer from the Dcache to the registers is two clock pulses while the latency from the Scache is 8–10 clock pulses. Both the Dcache and the Scache can be accessed at two loads per clock period and 1 store per clock period. This means that the peak bandwidth that is available on-chip is 4.8 GB/s for loads (300 MHz * 2 loads * 128/8 byte data bus, at a latency of two clock periods) and 2.4 GB/s for stores (one half the load bandwidth, as only one store can occur in a clock period). Table 11.3 gives the peak on-chip bandwidths for the different T3E systems.

Note that the Scache bandwidth for loads is the same as the Dcache bandwidth for loads, even though the Scache latency is 8–10 clock pulses. This is due to the fact that a single Scache line is two times as long as the Dcache line (which implies that two Dcache requests can be satisfied by one Scache line) and also, two Scache line fills an be taking place simultaneously from main memory. These two reasons combined make the latency of Scache the same as the Dcache (assuming an eight-clock-pulse latency for the Scache) and hence they have the same bandwidths.

The main memory is attached directly to each node and these local memories combined together make up the global memory. This physically distributed memory can be accessed by any node as logically shared. However, a given processor can access its local memory at a significantly faster rate than it can remotely located memory. Hence, the T3E is a non-uniform memory access (NUMA) architecture.

## Cache Coherence

The secondary cache does no pre-fetching, bringing in a cache line only when there is a miss at both levels of cache. A fetch also occurs in the Scache when there is a store request for a location not in the cache; this is

**Table 11.3** Peak On-Chip Bandwidths (GB/s)

| Type of access | Cray T3E | Cray T3E-900 | Cray T3E-1200 |
|---|---|---|---|
| Dcache load | 4.8 | 7.2 | 9.6 |
| Scache load | 4.8 | 7.2 | 9.6 |
| Dcache or Scache store | 2.4 | 3.6 | 4.8 |

a *write–allocate policy*. If modified, the cache line is tagged as *dirty*, and is written back to memory later; this is a *write-back mechanism*. The system uses the *write-back* mechanism for maintaining cache coherence, i.e., the dirty block is written back (updated) to memory only when necessary. This is done either by the Scache controller or by an external backmap system.

The *Scache controller* writes back a dirty cache line when it needs to make room for a new cache line. The replacement policy is random.

A *backmap* is a memory unit that is used to note addresses of valid entries within a cache. An external backmap is used here to maintain cache coherence with the local memory and the E-registers. The backmap filters memory references from remote nodes and probes the on-chip cache when necessary to invalidate lines or retrieve dirty data. The backmap consists of a set of tags that reflects the contents of the Scache. All of the E-register references first consult the backmap before referencing the local memory. If the data is in the Scache, the corresponding Scache line is *flushed* (written to memory and invalidated) and only then is the E-register operation allowed to continue.

### Support Circuitry

When reference misses occur in the Scache, they must go off chip to retrieve the necessary data. This means an increase in the latency and a reduction in the supply bandwidth. The support circuitry provides two mechanism that aid in the reduction of latency and increase the bandwidth for such references: streams and external registers (E-registers). Figure 11.15 gives a schematic of the support circuitry.

### Streams

Streams are a set of data buffers present on each node of the T3E, which lie betwen the Scache and main memory. There are six streams, each containing two 64-byte buffers. The stream buffer logic is designed to detect short- or unit-strided access patterns from the processor to fill Scache lines from local memory.

The stream detection hardware can keep track of the eight previous Scache misses. When it comes across two Scache misses that refer to two consecutive 64-byte blocks in main memory it activates a stream. The stream buffer logic then pre-fetches two consecutive Scache lines into this activated stream. Since there are six streams, each containing two 64-byte buffers, the stream buffers on the whole can hold up to twelve Scache lines. To aid the

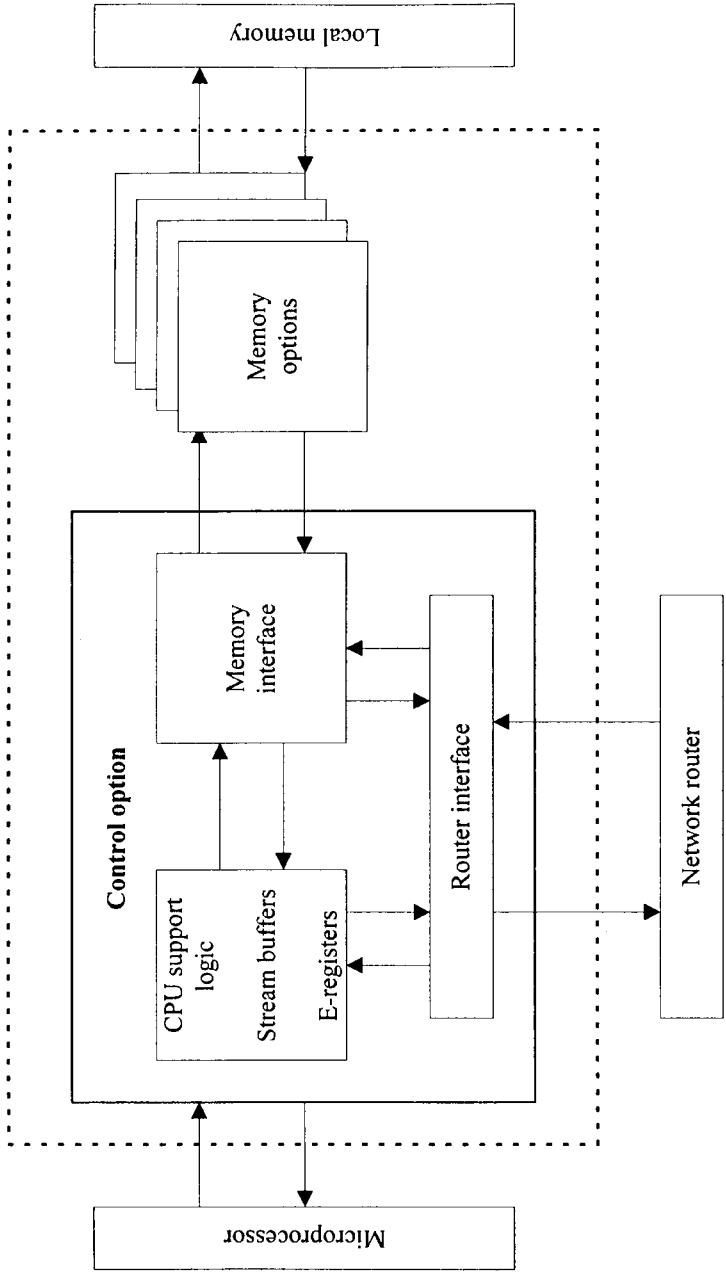



**Figure 11.15** Support circuitry for the T3E (Anderson et al., 1997)

stream buffer logic, the memory controllers also speculatively pre-fetch and buffer data for the active streams in units of four Scache lines. Note that the misses to consecutive memory blocks do not have to follow one another (they don't have to be contiguous in time); as long as there is a Scache miss that is spatially consecutive to one of the eight misses already stored by the detection hardware, a stream gets activated.

Without streams, there can be a maximum of two Scache line fills occurring simultaneously from main memory. This means that the transfer bandwidth from DRAM pages will be poor, more so if the accesses are interleaved. Further, if the accesses are for random single words, the efficiency would reduce even more as references are cache-line based (eight-word) and seven-eighths of the line brought in would not be used. Hence streams help increase the bandwidth to local memory, particularly for short-strided data accesses. Performance measurements taken on a T3E-900 with and without streams showed significant increases in memory bandwidth with streams activated. Streams improved memory bandwidth by a factor of 1.7 for cacheable stores, 2.8 for cacheable loads, and 2.2 for cacheable loads and stores. These measurements were taken for simple load/store operations on very long unit-stride vectors without performing any other operations on the data.

Streams could be more advantageously exploited if application code was developed with an eye toward maximizing stream lengths and generating six or fewer concurrent reference streams. The following are a few of the techniques that could be used to optimize streams:

- Splitting up loops such that the number of streams being accessed are limited
- Rearranging array dimensions to maximize inner loop trip count
- Rearranging the array dimensions to minimize the number of streams
- Grouping statements that use the same stream.

### External Registers

External registers (also known a E-registers) are memory-mapped, explicitly managed registers, and are the source and target of all internode communication in the T3E. They are also used for non-cached access to local memory, i.e., they bypass the cache hierarchy lying between the processor and main memory. Better than just a load/store mechanism for accessing remote memory, the E-register mechanism extends the physical address space of the microprocessor to cover the entire global physical address

space of the machine and also increase the pipelining attainable for global memory requests.

There are 512 E-registers available for the user and another 128 E-registers available solely for system use; they are manipulated via memory-mapped operations. These registers are accessed via the input/output space of the microprocessor (recall that the 39th bit of the physical address is used to distinguish between cacheable memory space and non-cacheable I/O space). E-registers are used mainly for:

- Direct loads and stores between E-registers and the processor registers
- Global E-register operations.

Direct loads and stores between the processor register and E-registers can take place only in the non-cached I/O space and are used to store operands into E-registers and load results from E-registers.

Global E-register operations are used to transfer data to/from global (remote or local) memory and perform messaging and atomic operation synchronization. The global operation to read memory into E-registers is called a *Get* and the operation to write E-register contents to memory is called a *Put*. There are two forms of Gets and Puts, single-word and vector. Vector Gets and Puts transfer a series of eight words separated by an arbitrary stride; they are very useful for increasing single-word load bandwidth. For example, row accesses in FORTRAN can be fetched into contiguous E-registers using strided Vector Gets. Then these blocks of E-registers can be loaded stride-one into the processor in cache-line-sized blocks (64-byte lines). This makes more efficient use of the bus when compared to the normal cache fills, since with normal caching only one word would be used of the eight 64-bit words fetched.

To load the contents of a global memory address into the processor, a Get followed by a *read* of the E-register content has to be performed. To modify the contents of a global memory address (remote node), a Put followed by a cache line update must take place.

If E-registers are used to transfer data to/from local memory, the cache is bypassed. To use the E-register operations for accessing memory local to the node, the processor number specified will have to be the local processor instead of a remote processor number during calls to the shared memory library. Or, a loop level directive CACHE_BYPASS can be used to suggest to the compiler that E-rgister operations need to be used for the local memory.

Access to these registers is implicitly synchronized by a set of state flags, one per E-register. The maximum data transfer rate between two nodes using Vector Gets or Puts is 480 MB/s; however, the E-register con-

trol logic overheads limit this figure somewhat, depending upon the operation.

### Network Router

The network router is responsible for actually transmitting packets of information between different nodes in the system. A packet carries with it a routing tag that gets assigned by the router at its source node. This tag is used as an index into the routing table at each node (see below) to look up the next hop in the message's route to its destination. When a node issues a request, the request packet sent from that node must carry with it the node's logical node number (see below), so the destination node can route the response back to that node. Figure 11.16 shows the block diagram of a network router for the T3E.

Each router chip has a "who am I" register that contains the *logical node number* (address) configured at that particular node. The *routing look-up table* is a 544-entry table used for looking up message-routing information; each routing-table entry provides the virtual to physical destination mapping and the path information to route to a particular destination. The logical node number of the destination is an index into this table. To support systems with more than 544 processors, the lower two bits of the logical node number are not used in the translation, i.e., only the higher-order bits are used as an index into the routing table. With this scheme, each line of the routing table would now map to a group of four neighboring nodes instead of just one node; hence, the table could support up to 2176 (4*544) nodes. Each entry in this table contains:

- The physical address of a node
- The deterministic path to that node, and
- Whether adaptive routing is allowed or not (see Section 0 below for more details on adaptive routing).

The processing element (PE) interface of the network, which is full duplex, performs the routing-table lookup at the source node and saves the return information at the destination-processing element. The arbiter is an intelligent piece of crossbar hardware which serves to balance the message load on the router's six external interfaces. It does this by monitoring the message load at each interface, and informing lightly loaded interfaces when they should take some of the message load from more heavily loaded interfaces; any transfers of this sort take place only with the arbiter's permission, and the messages travel via the crossbar hardware. In normal circumstances, when no interfaces are overloaded, an incoming message will leave the

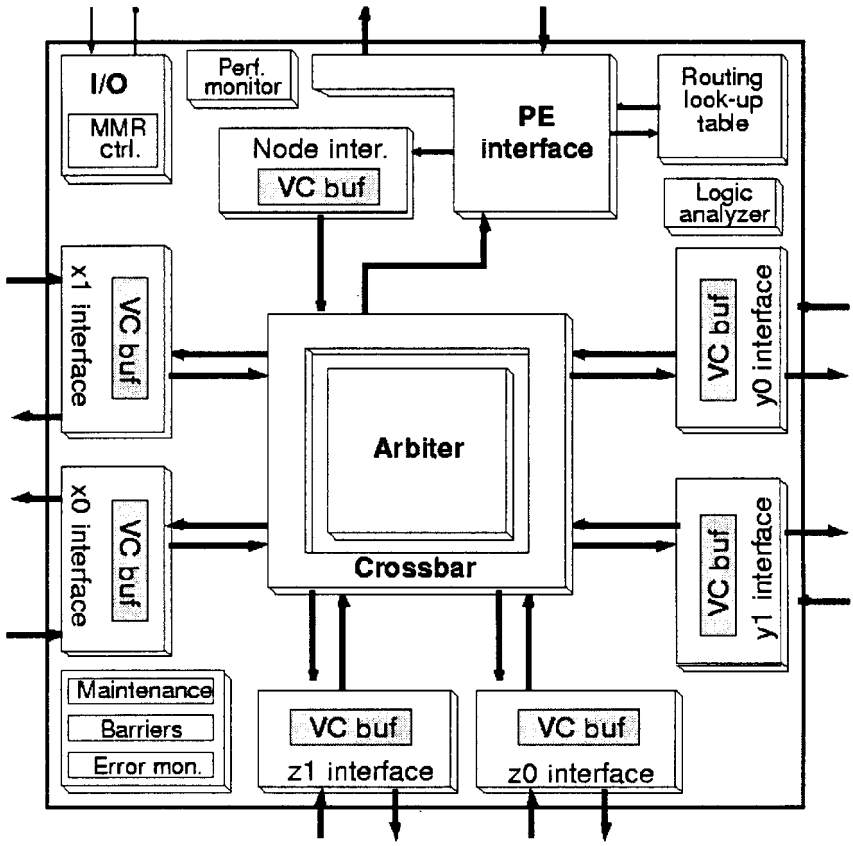



**Figure 11.16** Network router in a T3E (Scott, 1996)

router from the same interface by which it entered; this is possible because each interface has a *short-circuit path* incorporated into it. The arbiter informs each interface when it should or should not use its short-circuit path to route messages.

Message routing is done in direction-order. In dimension-order routing, the ordering of the dimensions are fixed, i.e., a packet routes in the correct direction (positive or negative) in the $X$ dimension, then the $Y$ dimension is traversed, followed by the $Z$ dimension. In direction-order routing, the packets traverse the six directions in a particular/fixed manner and the ordering of dimensions are not fixed. The direction ordering in the T3E is $+X + Y, +Z, -X, -Y, -Z$. This means that a packet can travel in a dimension in one direction and then later traverse that same dimension in

the opposite direction, which may lead to shorter alternate paths. This also means that network reconfiguration around faulty communication paths is made easier. The router also allows for what are known as free hops – an initial hop and/or a final hop. An *initial hop* is a hop that can be taken by a packet in one of the + directions (in any dimension) without adding an additional bit to its routing information. This can be accomplished by assigning an "initial direction" to the packet just before it enters the first router. A final hop can also be made in the $-Z$ direction once all the routes have been taken. These initial and final hops are useful when dealing with partial planes (see Section 0 below for a description of partial planes) and also when trying to route in a faulty/degraded system.

A packet is made up of one to ten flow control digits (*flits*) and each ***flit*** is mae up of five *phits*. Each phit is 14 bits wide (the width of the network link) implying that a flit is 70 bits, which is actually a 64-bit word plus six bits of control information. The routers operate at 75 MHz and transmit five phits over the network per clock period. This gives a network phit transmission rate of 375 MHz and a peak raw bandwidth of 600 MB/s (75 MHz * (70 bits – 6 bits of control information)). But after routing protocol overheads, the maximum bandwidth is closer to 500 MB/s.

## Processor Interconnection

"The interconnection network used in the T3E is a high-bandwidth, low-latency bidirectional 3D torus with fully adaptive routing, optimized virtual channel assignments, integrated barrier synchronization support and a fairly robust fault tolerance." It provides scalability up to 2048 nodes. Figure 11.17 shows a schematic of the network configuration.

Figure 11.17 shows a 4-ary 3-cube network, i.e., a radix-4 cube with three dimensions. This cube can have a total of $4^3 = 64$ nodes in all, and is symmetric. This interconnection network has ends along each direction of the three dimensions $X$, $Y$, and $Z$, connected together; this means that since ends are wrapped in each direction, more efficient resource utilization is possible, as the cube is typically symmetric; also, wire density is uniform. Packets can be transferred along these dimensions in either the positive or the negative direction, following direction-order routing, to reach the desired destination node.

The interconnect is bidirectional, hence locality can be better exploited as messages between near-neighbors can travel along the path provided instead of an indirect convoluted path; this reduces hops between nodes, and results in shorter connection paths. Also, a bidirectional interconnect is more fault tolerant, since if a communication path between two nodes fails,

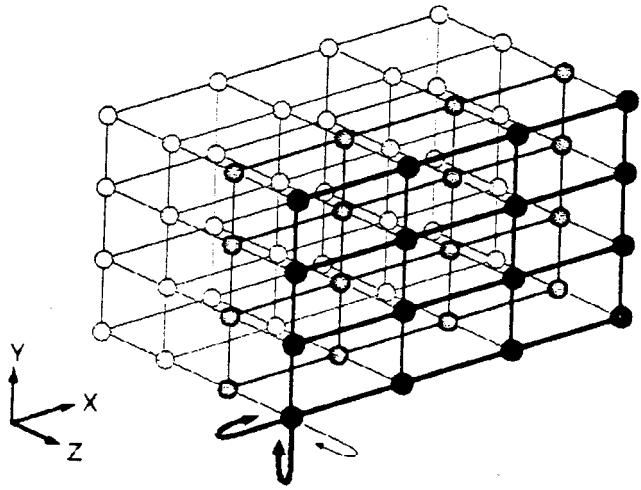



**Figure 11.17** 3D torus interconnect (Scott, 1996)

there are other paths that the message can pass through. Therefore, faulty paths might reduce the efficiency of message transfer, but the system will still perform.

Interprocessor data payload communication rates are 480 MB/s in every direction through the torus; the latency of the interconnect is about one microsecond on average. It allows for packets of the following types:

- Single-word (64-bit) or eight-word (64-byte) read and/or write requests
- Eight-word message packets
- Atomic memory operation packets
- Special packets used for system configuration and diagnostics.

The interconnect allows for the existence of planes in the $Z$ dimension that do not have a full set of nodes; these are called *partial planes*. A partial plane is useful for incorporating operating system nodes, spare nodes (for switching with faulty or redundant nodes), or in allowing a finer granularity of upgrade in system size. The free hops described earlier in Section 0 above is most useful in traversing the nodes of a partial plane. A packet originating from one of the nodes in the partial plane can make an initial hop in the $+Z$ direction and then follow the route from that node to the destination node. If on the other hand a destination node is in the partial plane, the packet can get routed from the source to the nearest $+Z$ neighbor of the destination node and then take a final hop in the $-Z$ direction to reach the destination.

Of course, these free hops need only be taken if the normal direction-ordered route does not exist from the source to the destination.

*Adaptive routing* is a concept that allows packets to use any minimal path to reach their destinations, i.e., any path that reduces the distance traveled. Adaptive routing is effective at improving throughput by allowing packets to be routed around local areas of congestion, or around a node that is faulty. The network provides adaptive routing through the use of virtual channels. Each physical channel is broken into five *virtual channels* (VCs), four for use in static routing and the fifth for adaptive routine. From any node in its path, a packet may take either the static virtual channel as determined by the routing algorithm or use the adaptive virtual channel in any ''profitable direction'' that gets it closer to its destination. A virtual channel used for adaptive routing is a little over twice the size of the largest packet used in the T3E. When certain communication paths are faulty and a packet must not be allowed to take an arbitrary minimal path, adaptive routing may be turned off, either by using the bit in the routing tag of a message packet or by letting the routing table turn off all adaptive routing traffic between any two nodes.

## I/O System

The T3E system performs all external I/O operations through a series of I/O controllers linked to a new technology called the GigaRing channel. At the core of this technology is an application-specific integrated circuit chip (ASIC) with may be linked with other circuitry to form an I/O node or ION. IONs must contain one ASIC, one I/O controller, and either a group of four PEs, or some external I/O device such as an FDDI, ATM, Ethernet, or SCSI device. Groups of IONs are linked in a ring topology to form a single I/O subsystem, which, in turn, can link with other subsystems. The whole network forms a GigaRing channel. Figure 11.18 shows a schematic of a sample GigaRing layout.

The ring topology provides great flexibility for distributed-system configuration. Large computers may connect to multiple GigaRing channels. Shared channels support shared peripherals and allow communication among multiple computers. Private channels allow a computer to access its own dedicated I/O system.

GigaRings support three primary functions:

- Direct memory access (DMA) transfers
- Messaging
- Remote client maintenance.

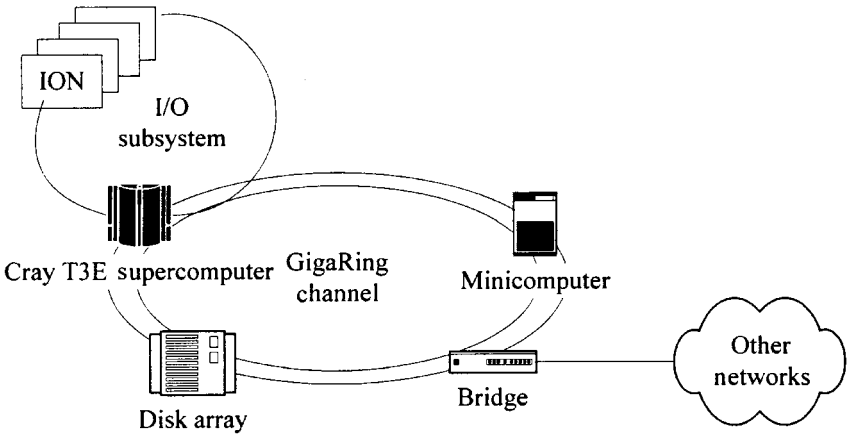



**Figure 11.18** GigaRing configuration (Scott, 1996)

DMA allows one client to directly read or write to another client's memory block. Messaging among peers can be accomplished without internal addressing information through a system sending out small packets and indicating that there should be no hardware response counterpart. And remote client maintenance is accomplished through control and diagnosis of memory-mapped registers viewable from client to client.

GigaRing channels are composed of three layers: physical, logical, and protocol. The *physical* layer provides the transport medium for the movement of the raw physical signals, while the *logical* layer provides an efficient packet delivery service. The ASIC chip combines these two layers and creates the *protocol* layer, which defines packets and communication procedures for client communication.

The *physical* GigaRing architecture is implemented using a dual-ring design, with data in the two rings traveling in opposite directions. The counter rotating rings allow for hot-swapping and resiliency to fiber cuts. Each ION contains one ASIC chip which is in turn connected to a client via either a 32- or 64-bit data bus. Data is transferred between the client and the ASIC at a selectable clock speed from 40 to 125 MHz. Communication on the ring, however, is locked at 75 MHz with data transfers on both edges of the clock signal, resulting in a 150 MB/s per signal transmission rate.

The ASIC must deal with at least four different clock rates: the client clock rate, internal data clock rates, and the possibly differing rates on both of the incoming ring channels. To cope with the plesiochronous transmissions that occur on the ring channels, special slip symbols may be added or delected from the signals. This latency tolerance allows for external node

connections of up to 15 feet (5 m) of standard SCSI-2 cabling and over 200 m for fiber-optic connections.

The *logical* layer enforces a variable length packet that must contain at least a 16-bit header followed by an optional data payload of up to 256 bytes. Figure 11.19 shows the GigaRing logical packet format.

The structure of a packet is as follows:

- *TargetID* and *SourceID* (each 13 bits) identify the packet's target and source. Each node connected to a GigaRing channel or interconnected network of channels has a unique nodeID, for a max of 8182 nodes.
- The *D* bit specifies delta (as opposed to absolute) addressing, which is useful for channel configuration and maintenance activities. If this bit is set, the TargetID represents a hop count rather than a node-ID.
- *Command* (13 bits) specifies the packet type, payload length (if appropriate) and error status. Packets are either requests or responses. Most GigaRing traffic consists of transactions, in which one client sends a request packet to another, and the other returns a response packet.
- *Flow* (5 bits) is used by the GigaRing node chip internally for flow control and error detection.
- *Sequence* (13 bits) allows clients to identify or order packets.
- *Control* (5 bits) allows clients to specify routing direction (either ring direction, or "don't care"), packet priority status, and packet ordering if desired.
- *C* bit marks corrupt packets, which can be delivered to the client for diagnostic purposes.
- *Address_High* and *Address_Low* compose a 64-bit internal address (64-bit word).

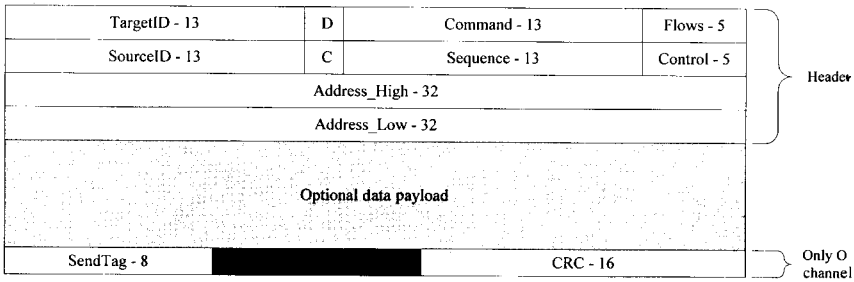



**Figure 11.19** Packet format (Scott, 1996)

Further error detection is accomplished through a 16-bit cyclic redundancy check word (CRC), as well as an extra parity bit per broadcast symbol.

The logical layer takes care of message balancing in the GigaRing channel. When the client is first connected to an ASIC chip, the client and ASIC chip exchange a series of messages that indicate the send and receive buffer sizes of each participant. This prevents a message buffer overflow of signals sent from other clients. If the received buffers do fill up on the ASIC chip, the message is *busied* and sent back to the original sender to indicate that the message was not received. The ASIC chip itself receives, parses, and either passes a signal down the ring, or moves the signal into the received buffer (when the buffer is not full) and awaits the clearing of the received buffer by the local client. The ASIC chip's parser can also automatically sends an echo packet back to the sending client when a signal is moved into the received buffer. Once a message is sent from a client into the ASIC chip, it is moved from the active send buffers onto the ring, and a backup copy of the message is stored locally until the corresponding echo bit is received. The ASIC chip then enters a 'recover phase' where it inhibits all downstream nodes from transmitting onto the ring until it can clear the contents of its bypass buffer (which may have been filling up during its own transmission).

A four-state finite-state machine associated with the receive buffers guarantee that, periodically, the buffers accept only the oldest set of busied packets – while refusing newer packets to prevent any packets from being busied indefinitely. Finally, the GigaRing channel accommodates differing speed clients through an adaptive congestion control mechanism, where the sender keeps track of the number of undelivered requests, as well as the number of unanswered requests through counting the response packets and keeping the sent vs. unanswered ratios within software controllable bounds. The logical level also provides some channel maintenance facilities such as configurable memory mapped registers (MMRs). Through the remote manipulation of these registers, a node can be tuned and debugged, as well as enabling other networking fault tolerance mechanisms such as ring masking and folding. Ring masking is simply the disabling of one of the two physical rings and forcing all transmits to occur on the one active ring. This technique is useful when one of the two ring cables is damaged or non-responsive. Ring folding is the physical connection of the two physical rings within the ASIC chip itself. This mechanism allows the hot-swapping and addition of nodes without service interruption. Figure 11.20 shows the schematic of the GigaRing ASIC chip, and Fig. 11.21 depicts the GigaRing masking and folding techniques.

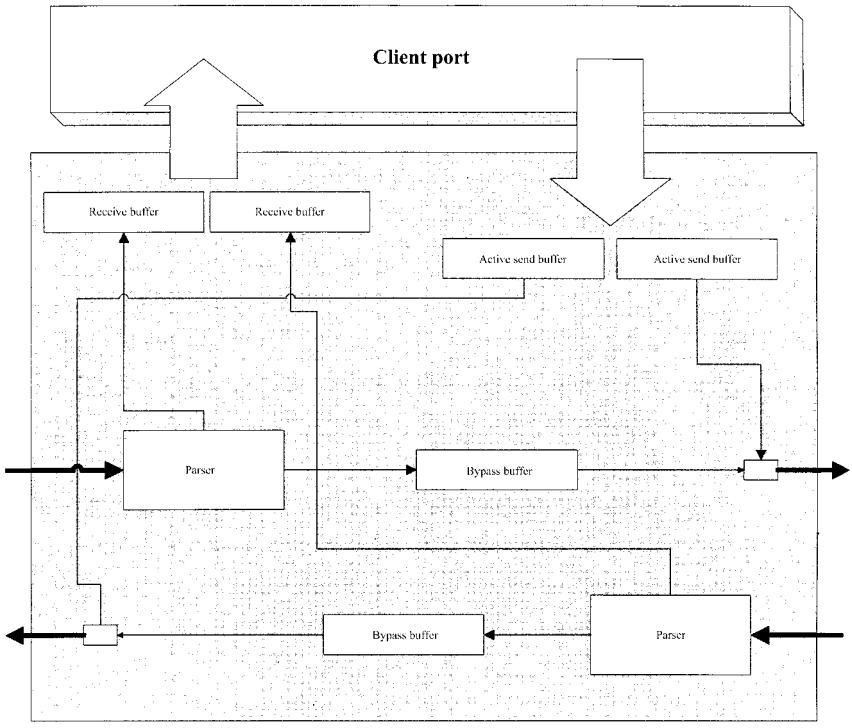



**Figure 11.20** GigaRing ASIC chip (Scott, 1996)

The protocol level on the GigaRing channel is pretty simplistic. A sender can either send a message in 256K packets from peer to peer, or arrange for a DMA memory copy. In a DMA transfer, the sender simply initiates the transfer, and hardware interrupts it when the transfer is complete.

## Operating System

To fully take advantage of the MIMD design of the Cray T3E, several software issues must be addressed. The operating system must efficiently utilize all the resources available, and efficient software tools must be provided to the developer to maximize efficiency. In response to the first issue, Cray provides the UNICOS/mk operating system with the Cray T3E. This microkernel-based OS provides basic hardware abstraction, memory management, thread scheduling and interprocess communication between the

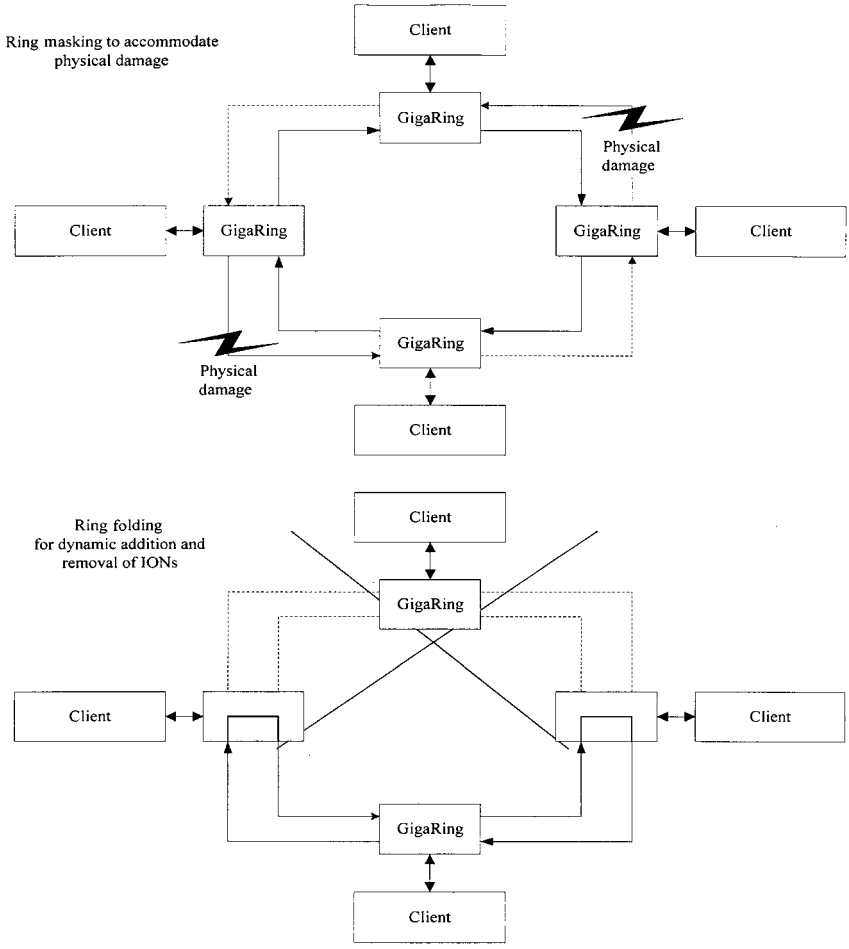



**Figure 11.21** Gigaring folding and masking techniques (Scott, 1996)

processes. Each PE hosts a copy of the UNICOS/mk OS server locally to handle local software resource concerns while roughly one out of every 16 processing elements is set aside by the OS to manage global operating system services in the form of "servers". A sample listing of servers is shown in Table 11.4

UNICOS/mk supports 32 and 64-bit addressing and file system extensions that support large data files and file systems. These extensions include disk striping and data caching capabilities, large file system support, and logical volumes that span multiple devices.

**Table 11.4** Sample Listing of Servers (Haataja and Savolainen, 1998)

| Server | Function |
|---|---|
| Process management (PM) | Manages processes local to its PE |
| Global process manager (GPM) | Manages lists of know processes |
| Global resource manager (GRM) | Tracks and allocates resource management |
| Configuration server (CS) | Processes requests for system configuration data |
| File server | Provides file system services |
| File server assistant | Provides file system services locally |

### Software Support

Cray also supplies a series of specialized tools for software development. Among these are several message passing and parallel communication libraries, and useful scientific and mathematical libraries that efficiently utilize the distributed nature of the T3E. These libraries may be accessed through any one of the supported programming languages (FORTRAN, C, C++, and HPF).

The Cray scientific libraries contain a collection of commonly used mathematical subroutines to solve specific problems efficiently. They include routines to compute simple matrix and vector operations such as addition and multiplication as well as solving more computationally intensive operations such as finding eigenvalues. Cray's Libsci also contains some communications libraries for working with individual portions of large distributed matrices and vectors.

Parallel software communication in the T3E-UNICOS/mk environment is accomplished through one of three special libraries: Message Passing Interface (MPI), Parallel Virtual Machine (PVM), and the Cray specific SHMEM (shared memory). While MPI and PVM are the most widely known of the interprocess communication libraries, SHMEM, Cray's Shared Memory Library, is worth evaluating. SHMEM was designed to allow low level manipulation of the T3E's E-registers and interconnection networks for an efficient method of interprocess communication. Care must be taken with SHMEM to ensure that data integrity is not compromised, as no data coherence mechanisms are included. What SHMEM does is to allow extremely efficient one-sided reads and to write directly to the memory of another PE, without accruing the overhead of a more data safe library such as MPI. The drawback is twofold: the programmer must take special

precaution not to destructively overwrite another PE's data, and the SHMEM library is not portable to other machines.

### Performance Analysis Tools

Performance analysis in a high-performance computing environment is essential to guarantee the most efficient usage of time, which is always at a premium. Combine this need for efficiency with the difficult to predict nature of parallel programming and the need for a strong performance analysis toolkit becomes more than apparent. Cray provides two basic tools with the T3E: Apprentice, and PAT.

Apprentice is a post-execution performance analysis tool for message passing programs which analyzes summary information collected at runtime via an instrumentation of the source program. The programmer selects a compiler switch at compile time which causes the compiler to insert special instrumentation into the executable (after optimization) and allow the program to collect the following information:

- Execution time
- Number of floating point, integer, and load/store operations
- Instrumentation overhead
- Subroutine execution time, and number of calls.

This data is combined into a runtime information file that is then analyzed by Apprentice, which generates and displays bar-charts of the information. This information is useful in determining the general behavior of the program, as well as targeting which subroutines are good candidates for further optimization. While Apprentice does not evaluate the hardware performance counters present on the Alpha processor, it does interpolate useful hardware performance information based on comparing 'best guess' performance comparisons.

Performance Analysis Tool (PAT) is Cray's more low-level analysis tool. In order to use PAT, the programmer must simply re-link his existing object code with the PAT libraries and execute the program. PAT uses strobe sampling to gather time estimates on how much time a program's execution thread remains within specified subroutines. PAT also utilizes the Alpha processors' built in hardware counters to find the exact number of floating-point and integer load/store operations as well as the number of cache misses. PAT's unique ability to instrument object code can allow the programmer to trace the program's entrance into and exit from external library to determine if the performance problems are occurring within his code or not. The major drawback is the requirement that the programmer

know the target function names as they occur in the object code, which can be confusing in the case of C++'s name mangling.

Apprentice and PAT allow the programmer a bird's-eye view into the inner workings of his/her program to more effectively target certain performance 'hot-spots' for optimization. As an illustration to the importance of using these tools, we submit the following scenario.

A programmer is developing a program that performs multiple distributed matrix element calculations. He compiles his code and includes Apprentice instrumentation. After executing his program on some test data, Apprentice observes that the program is spending the majority of its time in one specific function. The programmer then relinks his code with the PAT libraries and targets PAT's performance evaluation on this specific program. After re-executing his program on the same data, PAT reveals that about 50% of the program's execution time is spent in the following seemingly innocent loop:

```
for(I = 0; I < 1000; I++)
{
a[I] = b[I]*c[I];
t = d[I] + a[I];
e[I] = f[I] + T*g[I];
h[I] = h[I] + e[I];
}
```

PAT also reveals that the reason for the performance degradation is the large number of cache misses. After carefully evaluating the loop, the programmer realizes that in order to more effectively access the cache, the loop must access cache in strips of exactly 256 bytes. He also realizes that the arrays can be resized to more efficiently take advantages of streams. After making the above changes to the loop, further performance profiling indicates that performance was increased by 40% within this critical element. Given the distributed nature of this program, a 40% increase in performance at each PE could equate to a large increase in overall performance.

## 11.3 DATA-FLOW ARCHITECTURES

Data-flow architectures tend to maximize the concurrency of operations (parallelism) by breaking the processing activity into sets of the most primitive operations possible. Further, the computations in a data-flow machine are data-driven. That is, an operation is performed as and when its operands are available. This is unlike the machines we have described so

far, where the required data are gathered when an instruction needs them. The sequence of operations in a data-flow machine obey the precedence constraint imposed by the algorithm used rather than by the control statements in the program. A data-flow architecture assumes that a number of functional units are available, that as many of these functional units as possible are invoked at any given time, and these functional units are purely functional in the sense that they induce no side effects on either the data or the computation sequence.

The data-flow diagram of Fig. 11.22 shows the computation of the roots of a quadratic equation. Assuming that $a, b$, and $c$ values are available, $(-b)$, $(b^2)$ $(ac)$, and $(2a)$ can be computed immediately, followed by the computation of $(4ac)$, $(b^2 - 4ac)$, and $\sqrt{b^2 - 4ac}$), in that order. After this, $(-b + \sqrt{(b^2 - 4ac)}$ and $(-b - \sqrt{b^2 - 4ac})$ can be simultaneously computed followed by the simultaneous computation of the two roots. Note that the only requirement is that the operands be ready before an operation can be invoked. No other time or sequence constraints are imposed.

Figure 11.23 shows a schematic view of a data-flow machine. The machine memory consists of a series of cells where each cell contains an opcode and two operands. When both operands are ready, the cell is presented to the arbitration network that assigns the cell to either a functional unit (for operations) or a decision unit (for predicates). The outputs of functional units are presented to the distribution network, which stores the result in appropriate cells as directed by the control network. A very high throughput can be achieved if the algorithms are represented with the maximum degree of concurrency possible and the three networks of the processor are designed to bring about fast communication between the memory, functional, and decision units.

Two experimental data-flow machines (at the University of Utah and in Toulouse, France) have been built. The data-flow project at the Massachusetts Institute of Technology has concentrated on the design of languages and representation techniques and feasibility evaluation of data-flow concepts through simulation.

## 11.4 COMPUTER NETWORKS AND DISTRIBUTED PROCESSING

In an MIMD, several processors are connected with each other and the shared memory through the interconnection network, in a tightly coupled manner. The shared-memory interconnection of two or more computers discussed in Chapter 6 is another example of tightly coupled processors. A computer network, on the other hand, is a set of computer systems loosely

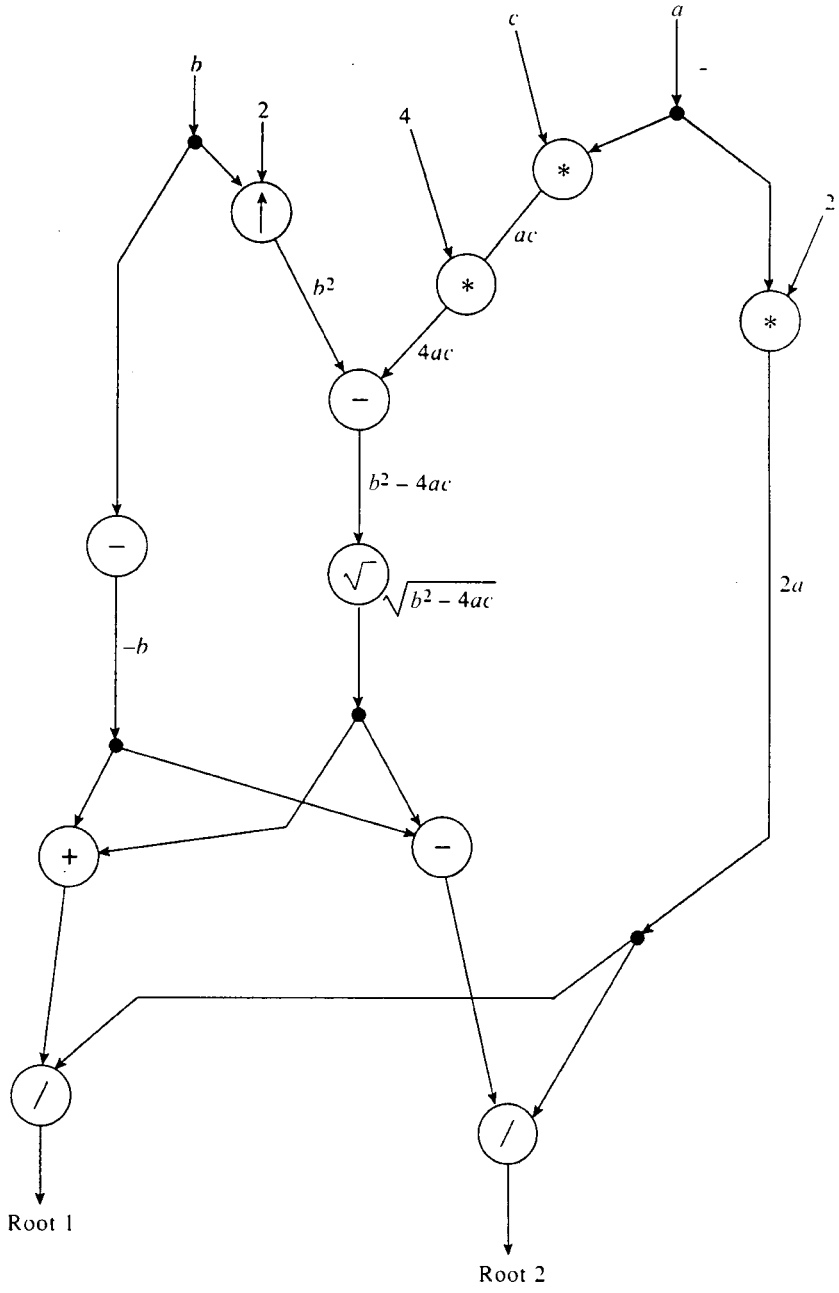



**Figure 11.22** A data-flow graph

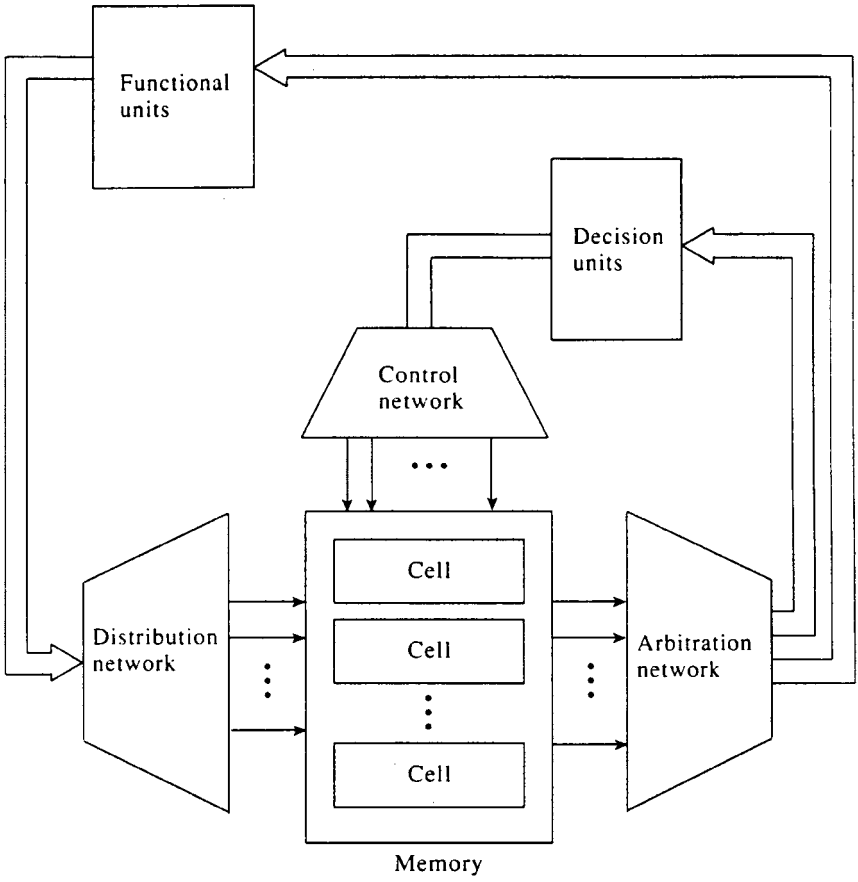



**Figure 11.23** A data-flow machine

coupled through a communication network. Figure 11.24 shows a model for a computer network. Each computer system in the network is a host. Each host is connected to a node in the communication network that transmits messages from and to the hosts connected to it. In a network, most of the processing is done by the local host at each site (node). If a resource (compiler, data base, etc.) is not available at the local host, a node calls upon a host that holds the rsource through the communication network.

With the introduction of microprocessors, local networks have become very popular because this system architecture provides a dedicated processor for local processing while providing the possibilities of sharing the resources with other nodes. Various networks have been built using large-scale

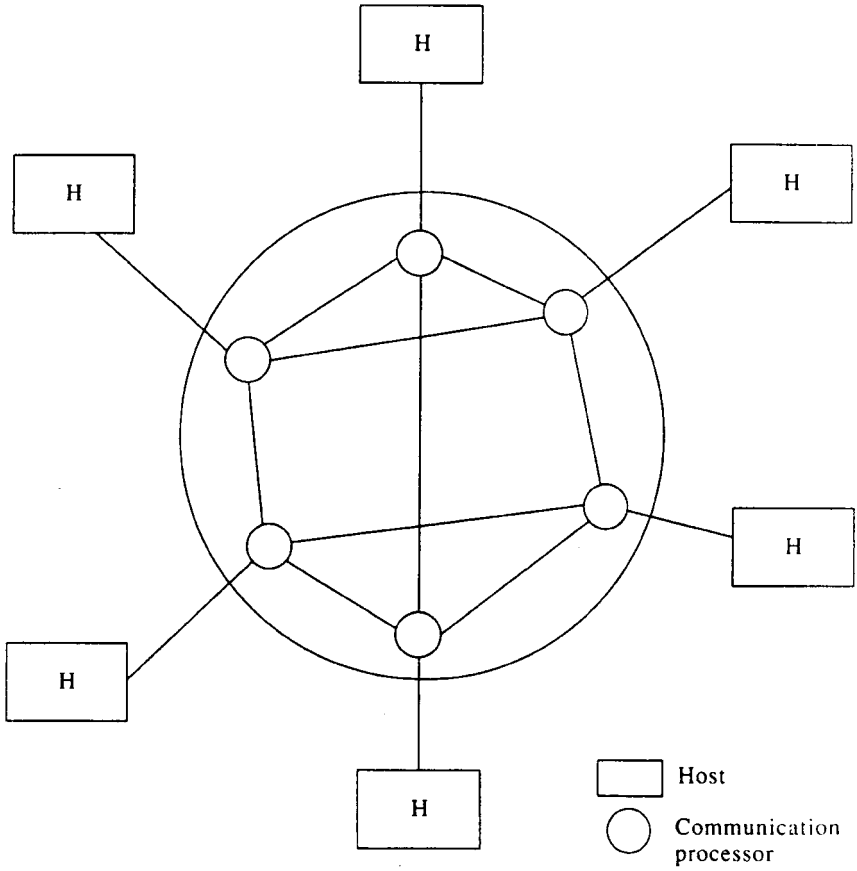



**Figure 11.24** A computer network

machines, minicomputers, and microcomputers. A number of topologies for communication networks exist.

In a computer network, if a node fails, the resources at that node are no longer available. If, on the other hand, each node in a network is made a general-purpose resource, the processing can continue even if a node fails, although at a reduced rate. A *distributed processing* system is one in which the processing, data, and control are distributed. That is, there are several general-purpose processors distributed geographically; no one processor will be a master controller at any time and there is no central data base; rather it is a combination of all the subdatabases on all the machines. Although no

such processing system exists to the author's knowledge, systems that adopt the distributed processing concepts to various degrees exist.

The advantages of distributed processing systems are

1. Efficient processing, since each node performs the processing for which it is most suited
2. Dynamic reconfiguration of the system architecture to suit the processing loads
3. Graceful degradation of the system in case of failure of a node, and redundancy if needed.

## 11.5 SUMMARY

This chapter provided brief descriptions of major trends in computer architecture. Developments in both hardware and software technologies have influenced computer design and architecture. VLSI fabrication densities have made it possible to implement more and more functions in hardware. VLSI technology has also made it possible to fabricate complex systems on a chip in a cost-effective manner, leading to the evolution of powerful desktop systems and RISC architectures. Availability of low-cost hardware and reduced communication costs have led to network-based architectures and distributed processing systems.

This chapter also provided an architecture classification scheme. As shown, practical machines do not fall neatly into one of the classifications in this scheme, but span several classifications according to the mode of operation.

Examples of representative supercomputers were provided. There are two schools of thought regarding the design of today's supercomputer architecture. The efforts in the USA are to build high-throughput numeric processors, such as CRAY, and network-based machines, such as the Connection Machine, while the efforts in Japan concentrate on building super numeric/symbolic processors that are based on concepts from the area of artificial intelligence. The Japanese efforts, known as fifth-generation system architectures, have not resulted in commercially feasible architectures.

The major aim of all the advanced architectures is to exploit the parallelism inherent in processing algorithms. The architecture parallelism spans from the fine-grain parallelism provided by data-flow architectures to coarse-grained parallelism provided by distributed systems. Newer technologies such as neural networks and optical computing structures are now being explored.

## REFERENCES

Alpha 21164 Microprocessor Hardware Reference Manual, Order Number EC-QP99C-TE, 1998, Compaq Computer Corporation.

Anderson, E. J. Brooks, and T. Hewitt, *The Benchmarker's Guide to Single-Processor Optimization for Cray T3E Systems*, Cray Research, June 1997.

Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. "The ILLIAC-IV Computer," *IEEE Transactions on Computers*, Vol. C-17, August 1968, pp. 746–757.

*Cray T3E White Papers*, http://www.sgi.com/t3e/performance.html.

Dennis, J. B. "First version of a data flow procedure language," in *Lecture Notes in Computer Science*. Berlin: Springer Verlag, 1974, pp. 362–376.

Dennis, J. B., and D. P. Misunas. "A preliminary data flow architecture for a basic data flow processor." *Proceedings of the Second Symposium on Computer Architecture*, 1975, pp. 126–376.

Haataja, J. and Savolainen, V. *Cray T3E Users Guide*, 1998, Center for Scientific Computing, Finland.

Hillis, W. D. *The Connection Machine*. Cambridge, MA.: MIT Press, 1985.

Hillis, W. D. "The Connection Machine," *Scientific American*, June 1987.

Hillis, W. D., and Steele, G. L. "Data parallel algorithms," *Communications of ACM*, December 1986.

Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Hingham, MA: Plenum, 1999.

Schaeffer, J. *High Performance Computing Systems and Applications*. Hingham, MA: Kluwer, 1988.

Scott, S., "The GigaRing Channel," *IEEE Micro*, vol. 16, no. 1, pp. 27–34, February 1996.

Shiva, S. G., *Pipelined and Parallel Computer Architectures*. New York: HarperCollins, 1996.

Scott, S., and G. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus" (slides) 1996. http://reality.sgi.com/sls_craypark/Papers/hot_icn96.html.

Silicon Graphics Computer Solutions, *Cray Hardware Solutions: Product Brochure*, 1998.

Steele, G. L., and Hillis, W. D. "Connection Machine Lisp: fine-grained parallel symbolic processing," *Proceedings of 1986 Conf. on Lisp and Functional Programming*. Cambridge, MA., August 1986, pp. 279–297.

Tucker, L. W. "Architecture and applications of the Connection Machine," *IEEE Computer*, August 1988.

Veen, A. H. "Data-flow machine architecture," *ACM Computing Surveys*, Vol. 18, No. 4, December 1986, pp. 365–396.

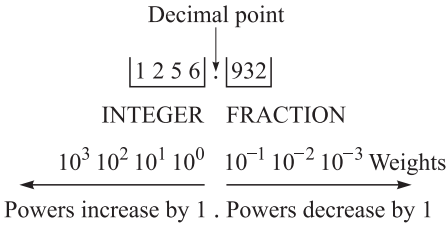
# Appendix A
## Number Systems and Codes

Number systems and computer codes are reviewed in this appendix. The books listed as references provide further details on these topics.

A number system consists of a set of symbols called *digits* and a set of relations defining the addition (+), subtraction (−), multiplication (×), and division (÷) of the digits. The total number of digits is the *radix* (or *base*) of the number system. In the "positional notation" of a number, the *radix point* separates the integer portion of the number from its fraction portion and if there is no fraction portion, the radix point is also omitted from the representation. Further, in this notation, each position has a weight equivalent to a power of the base. The power starts with 0 and increases by 1 as we move each position to the left of the radix point, and decreases by 1 as we move to the right. A typical number in decimal system is shown in Example A.1.

---

**Example A.1**




This number also can be represented as the polynomial

$$1 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 9 \times 10^{-1} + 3 \times 10^{-2} + 2 \times 10^{-3}.$$

---

A general positional representation of a number $N$ is

$$N = (a_n \cdots a_3 a_2 a_1 a_0 \cdot a_{-1} a_{-2} a_{-3} \cdots a_{-m})_r. \tag{A.1}$$

Where $r$ is the radix of the number system, $a_{-1}$, $a_0$, $a_1$, $a_2$, etc. are digits such that $0 \leq a_i \leq (r-1)$ for all $i$; $a_n$ is the most significant digit (MSD) and $a_{-m}$ is the least significant digit (LSD). The polynomial representation of the above number is

$$N = \sum_{i=-m}^{n} a_i r^i. \tag{A.2}$$

The largest integer that can be represented with $n$ digits is $(r^n - 1)$. Table A.1 lists the first few numbers in various systems. We will discuss binary, octal, and hexadecimal number systems. Digital computers employ the binary system, but octal and hexadecimal systems are useful in representing binary information in a shorthand form.

## A.1 BINARY SYSTEM

In this system, radix ($r$) is 2 and the allowed *bi*nary digi*ts* (bits) and 0 and 1. A typical number if shown below in the positional notation.

---

**Example A.2**

$$\begin{array}{rcll} N = (11010 & \cdot & 1101)_2 & \\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0 & \cdot & 2^{-1}2^{-2}2^{-3}2^{-4} & \text{Weights} \\ 16\ 8\ 4\ 2\ 1 & \cdot & \tfrac{1}{2}\ \tfrac{1}{4}\ \tfrac{1}{8}\ \tfrac{1}{16} & \text{Weights in decimal} \\ \longleftarrow & & \longrightarrow & \end{array}$$

Weights double for each move to left from binary point.

Weights are halved for each move to right from binary point.

In polynomial form, the above number is

$$\begin{aligned} N &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &\quad + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\ &= 16 + 8 + 0 + 2 + 0 + \tfrac{1}{2} + \tfrac{1}{4} + 0 + \tfrac{1}{16} \text{ Decimal} \\ &= 26 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{16} \text{ Decimal} \\ &= (26\tfrac{13}{16})_{10} \end{aligned}$$

---

Thus, to convert a number from binary to decimal, we can simply accumulate the weights corresponding to each nonzero bit of the binary number.

**Table A.1** Number Systems

| Decimal $r = 10$ | Binary $r = 2$ | Ternary $r = 3$ | Quaternary $r = 4$ | Octal $r = 8$ | Hexadecimal $r = 16$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 | 2 | 2 |
| 3 | 11 | 10 | 3 | 3 | 3 |
| 4 | 100 | 11 | 10 | 4 | 4 |
| 5 | 101 | 12 | 11 | 5 | 5 |
| 6 | 110 | 20 | 12 | 6 | 6 |
| 7 | 111 | 21 | 13 | 7 | 7 |
| 8 | 1000 | 22 | 20 | 10 | 8 |
| 9 | 1001 | 100 | 21 | 11 | 9 |
| 10 | 1010 | 101 | 22 | 12 | A |
| 11 | 1011 | 102 | 23 | 13 | B |
| 12 | 1100 | 110 | 30 | 14 | C |
| 13 | 1101 | 111 | 31 | 15 | D |
| 14 | 1110 | 112 | 32 | 16 | E |
| 15 | 1111 | 120 | 33 | 17 | F |
| 16 | 10000 | 121 | 100 | 20 | 10 |
| 17 | 10001 | 122 | 101 | 21 | 11 |
| 18 | 10010 | 200 | 102 | 22 | 12 |
| 19 | 10011 | 201 | 103 | 23 | 13 |
| 20 | 10100 | 202 | 110 | 24 | 14 |

Each bit can take the value of a 0 or a 1. With 2 bits, we can derive four ($2^2$) distinct combinations: 00, 01, 10, and 11. The decimal values of these patterns range from 0 to 3 (i.e., $2^2 - 1$). In general, with $n$ bits it is possible to have $2^n$ combinations of 0s and 1s and these combinations, when viewed as binary numbers, take values from 0 to $(2^n - 1)_{10}$. Table A.2 shows some binary numbers. A procedure for writing all the combinations of 0s and 1s for any $n$ follows.

With $n$ bits, $2^n$ combinations are possible. The first combination contains $n$ 0s and the last contains $n$ 1s. To generate the other combinations, note that in Table A.2, in the bit position $i(0 \leq i \leq n - 1)$, the 0s and 1s alternate every $2^i$ rows as we move from row to row. The least significant bit (LSB) corresponds to $i = 0$ and the most significant bit (MSB) corresponds to $i = n - 1$. That is, starting with a 0 in the first row, in the LSB position ($i = 0$), the values of bits alternate every row ($2^i = 2^0 = 1$); the values in bit position 1 ($i = 1$), alternate every two rows, etc. This procedure is repeated through the MSB ($i = n - 1$) position, to generate all the $2^n$ combinations.

**Table A.2** Binary Numbers

| $n = 2$ | $n = 3$ | $n = 4$ |
|---|---|---|
| 1 0 | 2 1 0 | 3 2 1 0 ← Bit position |
| 00 | 000 | 0000 |
| 01 | 001 | 0001 |
| 10 | 010 | 0010 |
| 11 | 011 | 0011 |
| | 100 | 0100 |
| | 101 | 0101 |
| | 110 | 0110 |
| | 111 | 0111 |
| | | 1000 |
| | | 1001 |
| | | 1010 |
| | | 1011 |
| | | 1100 |
| | | 1101 |
| | | 1110 |
| | | 1111 |

## A.2 OCTAL SYSTEM

In this system, $r = 8$ and the allowed digits are 0, 1, 2, 3, 4, 5, 6, and 7. A typical number is shown below in positional notation.

**Example A.3**

$$N = (4\ 5\ 2\ 6\cdot\ 2\ 3)_8$$

$$8^3 8^2 8^1 8^0 \cdot 8^{-1} 8^{-2} \qquad \text{Weights}$$

$$= 4 \times 8^3 + 5 \times 8^2 + 2 \times 8^1 + 6 \times 8^0 + 2 \times 8^{-1} + 3 \times 8^{-2} \qquad \text{Polynomial form}$$

$$= 2048 + 320 + 16 + 6 + \tfrac{2}{8} + \tfrac{3}{64} \qquad \text{Decimal}$$

$$= (2390\tfrac{19}{64})_{10}$$

## A.3 HEXADECIMAL SYSTEM

In this system, $r = 16$, and the allowed digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Digits A through F correspond to decimal values 10 through 15. A typical number is shown in Example A.4.

---

**Example A.4**

$$N = (\mathrm{A\ 1\ F \cdot 1\ C})_{16}$$

$$16^2 16^1 16^0 \cdot 16^{-1} 16^{-2} \qquad \text{Weights}$$

$$= A \times 16^2 + 1 \times 16^1 + \mathrm{F} \times 16^0 + 1 \times 16^{-1} + \mathrm{C} \times 16^{-2} \longleftarrow$$

Polynomial form

$$= 10 \times 16^2 + 1 \times 16^1 + 15 \times 16^0 + 1 \times 16^{-1} + 12 \times 16{-}2$$

Decimal

$$= (2591\tfrac{28}{256})_{10}$$

---

## A.4 CONVERSION

To convert numbers from any system to decimal, we simply expand the given number as a polynomial and evaluate the polynomial using decimal arithmetic, as we have seen in Examples A.1–A.4.

To convert an integer number in decimal to any other system, we use the *radix divide* technique. This technique requires the following steps:

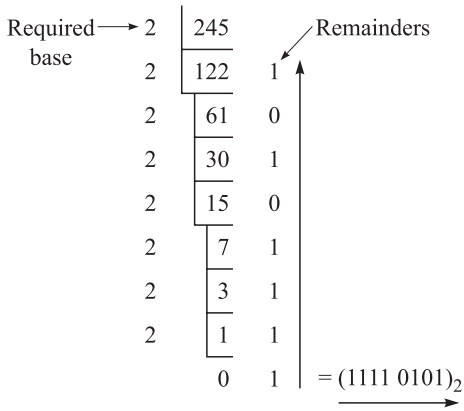
1. Divide the given number successively by the required base, noting the remainders at each step; the quotient at each step becomes the new dividend for subsequent division. Stop division when the quotient reaches 0.
2. Collect the remainders from each step (last to first) and place them left to right to form the required number.

This procedure is illustrated by the following examples.

---

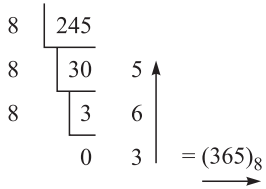
**Example A.5**

$(245)_{10} = (?)_2$ (i.e., convert $(245)_{10}$ to binary).



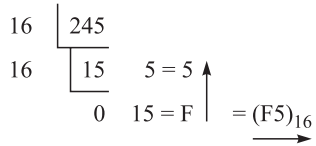

**Example A.6**

$(245)_{10} = (?)_8$.




**Example A.7**

$(245)_{10} = (?)_{16}$.




To convert a fraction in decimal to any other system, the *radix multiply* technique is used. The steps in this method are as follows:

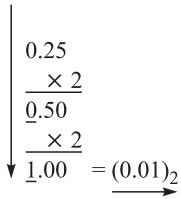
1. Successively multiply the given fraction by the required base, noting the integer portion of the product at each step. Use the fractional part of the product as the multiplicand for subsequent steps. Stop when the fraction either reaches 0 or recurs.
2. Collect the integer digits at each step from first to last and arrange them left to right.

It is not possible to represent a given decimal fraction in binary exactly if the radix multiplication process does not converge to 0. The accuracy, then, depends on the number of bits used to represent the fraction. Some examples follow.

---

**Example A.8**

$(0.250)_{10} = (?)_2$.

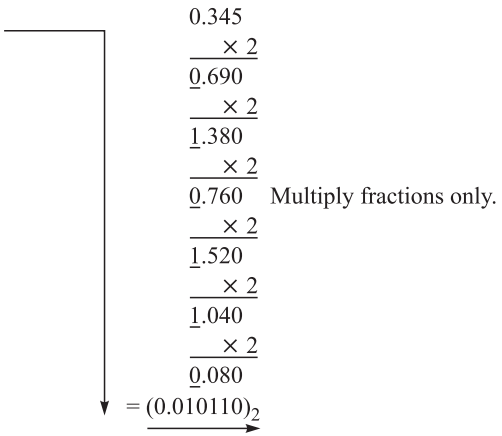



---

---

**Example A.9**

$(0.345)_{10} = (?)_2$.



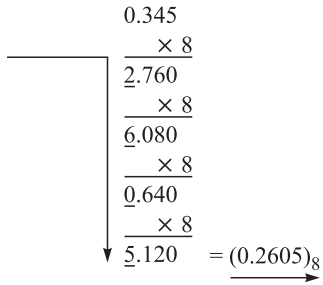

---

The fraction may never reach zero; stop when the required number of fraction digits are obtained, the fraction will not be exact.

---

**Example A.10**

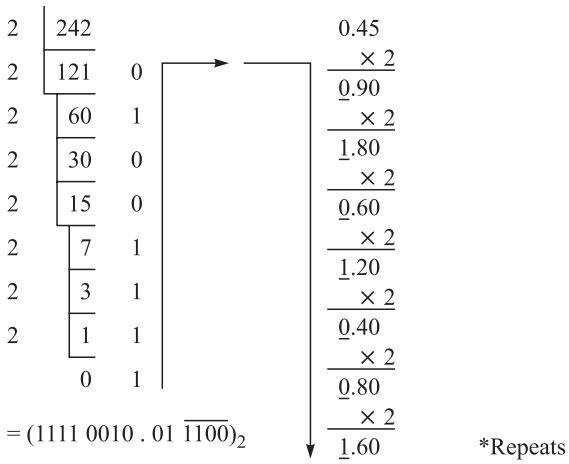
$(0.345)_{10} = (?)_8$.




---

**Example A.11**

$(242.45)_{10} = (?)_2$.




---

The radix divide and multiply algorithms are applicable to the conversion of numbers from any base to any other base. In converting a number from base $p$ to base $q$, the number in base $p$ is divided (or multiplied) by $q$ in

*base p arithmetic*. These methods are convenient when $p$ is 10 because of our familiarity with decimal arithmetic. In general, it is easier to convert a base $p$ number to base $q(p \neq 10, q \neq 10)$ by first converting the number to decimal from base $p$ and then conveting this decimal number into base $q$; that is, $(N)_p \rightarrow (?)_{10} \rightarrow (?)_q$, as shown by Example A.12.

---

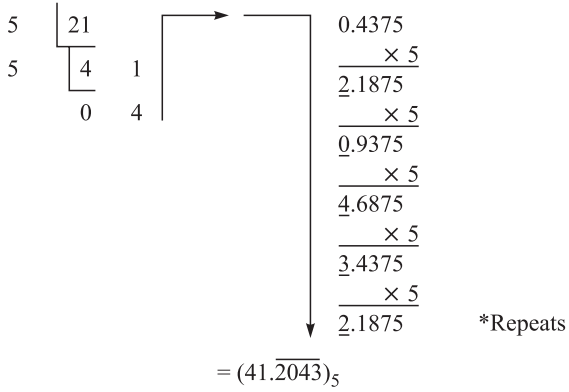
**Example A.12**

$$(25.34)_8 = (?)_5.$$

*Convert to base 10*

$$\begin{aligned}(25.34)_8 &= 2 \times 8^1 + 5 \times 8^0 + 3 \times 8^{-1} + 4 \times 8^{-2} \text{ Decimal} \\ &= 16 + 5 + \tfrac{3}{8} + \tfrac{4}{64} \text{ Decimal} \\ &= (21\tfrac{28}{64})_{10} \\ &= (21.4375)_{10}\end{aligned}$$

*Convert to base 5*




---

### A.4.1 Base $2^k$ Conversion

When converting a number from base $p$ to base $q$, if $p$ and $q$ are both powers of 2 an easier conversion procedure may be used. Here, the number in base $p$ is first converted into binary, which number is then converted into base $q$. This is called *base $2^k$ conversion*.

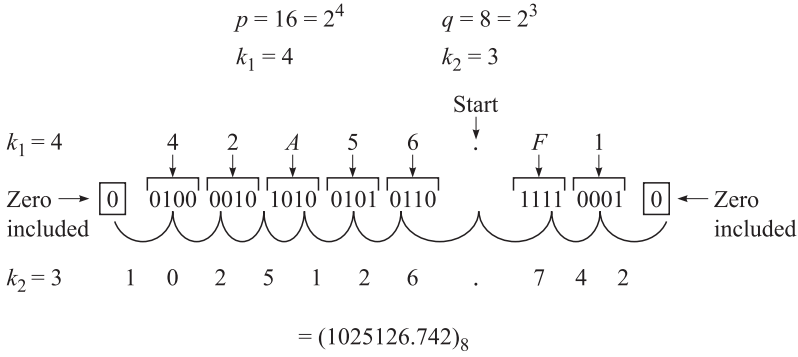
Let $p = 2^{k_1}$. In base $2^k$ conversion, each digit of the number in base $p$ is first expanded into a $k_1$-bit binary number. The bits of the number thus

obtained are regrouped into groups of $k_2$ bits; each such group is equivalent to a digit in base $q$ ($q = 2^{k_2}$). The grouping starts at the radix point and proceeds in either direction. Some examples follow.

---

**Example A.13**

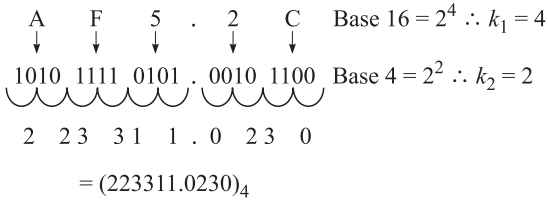
$$(4\ 2\ A\ 5\ 6 \cdot F\ 1)_{16} = (?)_8.$$




---

**Example A.14**

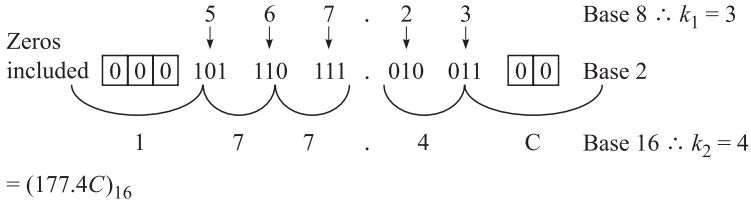
$$(\mathrm{AF5.2C})_{16} = (?)_4.$$




---

**Example A.15**

$$(567.23)_8 = (?)_{16}.$$




$= (177.4C)_{16}$

## A.5 ARITHMETIC

This section provides a reviewof binary, octal, and hexadecimal arithmetics. Arithmetic in any number system is similar to that in the decimal system. Binary arithmetic is easier to perform (and understand) than decimal arithmetic since only two digits are involved. Arithmetic in octal and hexadecimal systems is more complex and may require practice if we are unfamiliar with those systems.

### A.5.1 Binary Arithmetic

Table A.3 shows the rules for binary addition, subtraction, and multiplication.

**Table A.3** Binary Arithmetic

| A + B | A = 0 | A = 1 |
|---|---|---|
| B = 0 | 0 | 1 |
| B = 1 | 1 | 10 (CARRY SUM) |

(a) Addition

| A − B | A = 0 | A = 1 |
|---|---|---|
| B = 0 | 0 | 1 |
| B = 1 | 11 (BORROW DIFFERENCE) | 0 |

(b) Subtraction

| A × B | A = 0 | A = 1 |
|---|---|---|
| B = 0 | 0 | 0 |
| B = 1 | 0 | 1 |

(c) Multiplication

**Addition** From Table A.3(a), note that 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. Thus the addition of two 1s results in a SUM of 0 and a CARRY of 1. This CARRY is used in the addition of bits in the next most significant position in the addition of two binary numbers, as shown by Example A.16.

---

**Example A.16**

| 4 | 3 | 2 | 1 | 0 | Bit position |
|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | | CARRY | CARRY in the LSB position is 0. |
| | 1 | 0 | 1 | 1 | 0 | Augend | |
| + | 0 | 0 | 1 | 1 | 1 | Addend | |
| | 1 | 1 | 1 | 0 | 1 | SUM | |

Here, bits in the LSB (bit position 0) position are first added to generate a SUM bit of 1 and a CARRY bit of 0. This CARRY is used in the addition of bits in position 1. This process is repeated through the MSB (bit position 4).

---

In general, addition of two $n$-bit numbers results in a SUM containing a maximum of $(n + 1)$ bits. If the arithmetic has to be confined to $n$ bits, care should be taken to see that the two numbers are small enough so that their SUM does not exceed $n$ bits. Another example of addition follows.

---

**Example A.17**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | | CARRY |
| | 1 | 0 | 1 | 1 | 0 | 1 | 0 | Augend |
| | 0 | 1 | 1 | 0 | 1 | 1 | 1 | Addend |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | SUM |

---

## Subtraction

From Table A.3(b), $0 - 0 = 0$, $1 - 0 = 1$, $1 - 1 = 0$, and $0 - 1 = 1$ with a BORROW of 1. Subtraction of two binary numbers is performed starting from LSB toward MSB one stage at a time. Subtracting a 1 from a 0 results

in a 1 with a BORROW of 1 from the next (more significant) stage. Some examples follow.

---

**Example A.18**

| 5 | 4 | 3 | 2 | 1 | 0 | Bit position |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| | 0 | | | 0 | | |
| | ~~1~~ | 0 | 1 | ~~1~~ | 0 | 1 | Minuend |
| − | 0 | 1 | 1 | 0 | 1 | 0 | Subtrahend |
| | | 1 | 0 | 0 | 1 | 1 | DIFFERENCE |

Bit position 1 requires a borrow from bit position 2. Due to this borrow, minuend bit 2 is a 0. The subtraction continues through the MSB.

---

**Example A.19**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Bit position |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | | | 0 | | | |
| | ~~1~~ | ~~1~~ | ~~0~~ | ~~0~~ | ~~1~~ | 0 | 1 | 1 | Minuend |
| | | | 1 | 1 | | | | |
| − | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Subtrahend |
| | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | DIFFERENCE |

Bit 2 requires a borrow from bit 3; after this borrow, minuend bit 3 is 0. Then, bit 3 requires a borrow. Since bits 4 and 5 of the minuend are 0s, borrowing is from bit 6. In this process, the intermediate minuend bits 4 and 5 each attain a value of 1 (compare this with the decimal subtraction). The subtraction continues through the MSB.

---

### Multiplication

The multiplication is similar to decimal multiplication. From Table A.3(c), $0 \times 0 = 0, 0 \times 1 = 0, 1 \times 0 = 0$, and $1 \times 1 = 1$. An example follows.

**Example A.20**

```
1011          Multiplicand
 ×1100        Multiplier          ┌── Multiplier bits
                                  ↓
  0000                  (1011) × 0
 0000                   (1011) × 0
1011                    (1011) × 1
1011                    (1011) × 1
10000100      PRODUCT
```

In general, the PRODUCT of two $n$-bit numbers is $(2n)$ bits long. To multiply two $n$-bit numbers A and B, where B = $(b_{n-1}b_{n-2}\cdots b_1b_0)$, the following procedure can be used:

1. Start with a $2n$-bit PRODUCT with a zero value (all 0s).
2. For each $b_i(0 \leq i \leq n-1) \neq 0$: shift A, $i$ positions to the left and add to the PRODUCT.

Thus, binary multiplication can be performed by repeated shifting and addition of the multiplicand to the partial product, to obtain the PRODUCT.

## Division

Division is the repeated subtraction of the divisor from the dividend. The longhand procedure of decimal division can also be used in binary.

**Example A.21**

110101 ÷ 111.

```
          0111      Quotient
111 ⟌ 110,101                 110  < 111 ∴ q1 = 0
     −000
      1101                    1101 > 111 ∴ q2 = 1
     −111
      1100                    1100 > 111 ∴ q3 = 1
     −111
      1011                    1011 > 111 ∴ q4 = 1
     −111
       100      Remainder
```

Note that the division is repeated subtraction of the divisor from the dividend.

References listed at the end of this appendix provide the details of several other binary multiplication and division algorithms. Hardware implementation of binary addition and subtraction is discussed in Chapter 1 of this book and multiplication and division algorithms are given in Chapter 10.

## Shifting

In general, shifting a base $r$ number left by one position (and inserting a 0 into the vacant LSD position) is equivalent to multiplying the number by $r$; and shifting the number right by one position (inserting a 0 into the vacant MSD position) is equivalent to dividing the number by $r$.

In the binary system, each left shift multiplies the number by 2 and each right shift divides the number by 2, as shown by Example A.22.

---

**Example A.22**

| | | Binary | | | Decimal | |
|---|---|---|---|---|---|---|
| $N$ | | 01011.11 | | | 11¾ | |
| $2 \cdot N$ | | 10111.1 | 0 | INSERT | 23½ | |
| $N \div 2$ | INSERT 0 | 0101.11 | | | 5¾ | (Inaccurate, since only 2-bit accuracy is retained.) |

---

If the MSB of an $n$-bit number is not 0, shifting left would result in a number larger than the magnitude that can be accommodated in $n$ bits and the 1 shifted out of MSB cannot be discarded; if nonzero bits shifted out of LSB bit during a right shift are discarded, the accuracy is lost.

### A.5.2 Octal Arithmetic

Table A.4 shows the addition and multiplication tables for octal arithmetic. The following examples illustrate the four arithmetic operations and their similarity to decimal arithmetic.

**Table A.4** Octal Arithmetic

| | A + B | A 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 |
| | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 |
| B | 3 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| | 4 | 4 | 5 | 6 | 7 | 10 | 11 | 12 | 13 |
| | 5 | 5 | 6 | 7 | 10 | 11 | 12 | 13 | 14 |
| | 6 | 6 | 7 | 10 | 11 | 12 | 13 | 14 | 15 |
| | 7 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

(a) Addition

| | A × B | A 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 2 | 0 | 2 | 4 | 6 | 10 | 12 | 14 | 16 |
| B | 3 | 0 | 3 | 6 | 11 | 14 | 17 | 22 | 25 |
| | 4 | 0 | 4 | 10 | 14 | 20 | 24 | 30 | 34 |
| | 5 | 0 | 5 | 12 | 17 | 24 | 31 | 36 | 43 |
| | 6 | 0 | 6 | 14 | 22 | 30 | 36 | 44 | 52 |
| | 7 | 0 | 7 | 16 | 25 | 34 | 43 | 52 | 61 |

(b) Multiplication

**Example A.23** *Addition.*

| | | |
|---|---|---|
| 1 1 1 | ← Carries | Scratch pad |
| 1 4 7 6 | | 6 |
| + 3 5 5 4 | | +4 |
| 5 2 5 2 | SUM | $(10)_{10} = (12)_8$ |
| | | 1 |
| | | +7 |
| | | +5 |
| | | $(13)_{10} = (15)_8$ |
| | | 1 |
| | | +4 |
| | | +5 |
| | | $(10)_{10} = (12)_8$ |
| | | 1 |
| | | +1 |
| | | +3 |
| | | $(5)_{10} = (5)_8$ |

**Example A.24** *Subtraction.*

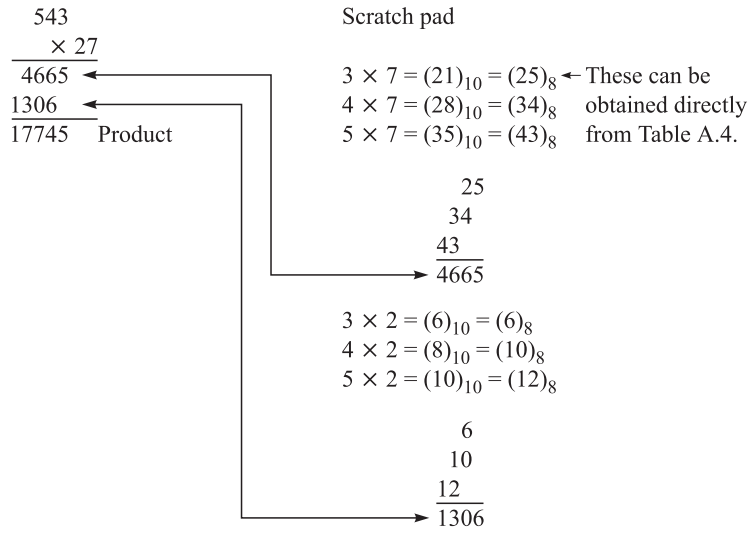
| | | | |
|---|---|---|---|
| 4 14 | Digit position 2 | | |
| ~~5~~ ~~4~~ 7 5 | Borrow from position 3. | | |
| −3 7 6 4 | ∴ | Octal | Decimal |
| 1 5 1 1 | | $14_8$ | 12 |
| | | $-7_8$ | −7 |
| | | $5_8$ | $5_{10} = 5_8$ |

**Example A.25**

$$\begin{array}{r} \phantom{-}3\ 7\ 7 \\ 5\ \not{4}\ \not{0}\ \not{0}\ 4\ 5 \\ -3\ 2\ 5\ 6\ 5\ 4 \\ \hline 2\ 1\ 2\ 1\ 7\ 1 \end{array}$$

The intermediate 0s become $(r - 1)$ or 7 when borrowed.

**Example A.26** *Multiplication.*




**Example A.27** *Division.* 543 ÷ 7.

```
    062
7 | 543
  | 0
  | 54
  | 52
  |  23
  |  16
  |   5
```

Use the multiplication table of Table A.4 to derive the quotient digit (by trial and error).

### A.5.3 Hexadecimal Arithmetic

Table A.5 provides the addition and multiplication tables for hexadecimal arithmetic. The following examples illustrate the arithmetic.

**Table A.5** Hexadecimal Arithmetic

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 8 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| A | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| B | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |
| C | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B |
| D | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C |
| E | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |
| F | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E |

(a) Addition

| × | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 2 | 0 | 2 | 4 | 6 | 8 | A | C | E | 10 | 12 | 14 | 16 | 18 | 1A | 1C | 1E |
| 3 | 0 | 3 | 6 | 9 | C | F | 12 | 15 | 18 | 1B | 1E | 21 | 24 | 27 | 2A | 2D |
| 4 | 0 | 4 | 8 | C | 10 | 14 | 18 | 1C | 20 | 24 | 28 | 2C | 30 | 34 | 38 | 3C |
| 5 | 0 | 5 | A | F | 14 | 19 | 1E | 23 | 28 | 2D | 32 | 37 | 3C | 41 | 46 | 4B |
| 6 | 0 | 6 | C | 12 | 18 | 1E | 24 | 2A | 30 | 36 | 3C | 42 | 48 | 4E | 54 | 5A |
| 7 | 0 | 7 | E | 15 | 1C | 23 | 2A | 31 | 38 | 3F | 46 | 4D | 54 | 5B | 62 | 69 |
| 8 | 0 | 8 | 10 | 18 | 20 | 28 | 30 | 38 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 |
| 9 | 0 | 9 | 12 | 1B | 24 | 2D | 36 | 3F | 48 | 51 | 5A | 63 | 6C | 75 | 7E | 87 |
| A | 0 | A | 14 | 1E | 28 | 32 | 3C | 46 | 50 | 5A | 64 | 6E | 78 | 82 | 8C | 96 |
| B | 0 | B | 16 | 21 | 2C | 37 | 42 | 4D | 58 | 63 | 6E | 79 | 84 | 8F | 9A | A5 |
| C | 0 | C | 18 | 24 | 30 | 3C | 48 | 54 | 60 | 6C | 78 | 84 | 90 | 9C | A8 | B4 |
| D | 0 | D | 1A | 27 | 34 | 41 | 4E | 5B | 68 | 75 | 82 | 8F | 9C | A9 | B6 | C3 |
| E | 0 | E | 1C | 2A | 38 | 46 | 54 | 62 | 70 | 7E | 8C | 9A | A8 | B6 | C4 | D2 |
| F | 0 | F | 1E | 2D | 3C | 4B | 5A | 69 | 78 | 87 | 96 | A5 | B4 | C3 | D2 | E1 |

(b) Multiplication

**Example A.28** *Addition.*

Scratch pad

Decimal

```
    1  1
  1 5 F C      C = 12
+ 2 4 5 D      D = 13
  3 A 5 9
```

16 | 25 = $(19)_{16}$ ← These can be obtained directly from Table A.5.

16 | 1 9 ↑

0 1

Decimal

1 = 1
F = 15
5 = 5
21 = $(15)_{16}$

**Example A.29** *Subtraction.*

Scratch pad

```
   1 13 15
     3
   ~~2~~ ~~4~~ ~~5~~ D  Minuend
-  1  5  F  C  Subtrahend
   0  E  6  1  Difference
```

Decimal

$(15)_{16}$ = 21
$-(F)_{16}$ = −15
6 = $(6)_{16}$
$(13)_{16}$ = 19
$-(5)_{16}$ = −5
14 = $(E)_{16}$

**Example A.30** *Multiplication.*

```
    1 E 4 A
        × FA2
    ---------
      1 2
    0 3 C 9 4      P1
+   1 2 E E 4      P2
+1 C 6 5 6     P3
  1 D 9 8 0 D 4
```

Scratch pad

```
Decimal                Hexadecimal
A × 2 = 20 =              1 4
4 × 2 =  8 =              0 8
E × 2 = 28 =            1 C
1 × 2 =  2 =            0 2
                        ---------
                        0 3 C 9 4 = P1

A × A = 100 =             6 4
4 × A =  40 =             2 8
E × A = 140 =           8 C
1 × A =  10 =           A
                        ---------
                        1 2 E E 4 = P2

A × F = 150 =             9 6
4 × F =  60 =             3 C
E × F = 210 =           D 2
1 × F =  15 =           D
                        ---------
                        1 C 6 5 6 = P2
```

**Example A.31** *Division.* 1 A F 3 ÷ E.

```
    0 1 E C   Quotient
E | 1 A F 3
  | 0
  | -------
  | 1 A
  |   E
  | -------
  |   C F
  |   C 4
  | -------
  |     B 3
  |     A 8
  | -------
        B   Remainder
```

## A.6 SIGN–MAGNITUDE REPRESENTATION

In this representation, the MSD of the number represents the sign. It is 0 for positive numbers and $(r-1)$ for negative numbers, where $r$ is the base of the number system. Some representation examples follow.

---

**Example A.32**

$$
\begin{aligned}
(+25)_{10} &= 0.0025 \\
(-25)_{10} &= 9.0025 \\
(+10)_{2} &= 0.0010 \\
(-10)_{2} &= 1.0010 \\
(+56)_{8} &= 0.0056 \\
(-56)_{8} &= 7.0056 \\
(+1\text{F})_{16} &= 0.001\text{F} \\
(-1\text{F})_{16} &= \text{F}.001\text{F} \\
&\quad\uparrow \\
&\quad\text{Sign, magnitude}
\end{aligned}
$$

All numbers are shown as five-digit numbers. In arithmetic with sign–magnitude numbers, the magnitude and sign are treated separately.

---

## A.7 COMPLEMENT NUMBER SYSTEMS

Consider the subtraction of a number A from a number B. This is equivalent to adding $(-\text{A})$ to B. The complement number system provides a convenient way of representing negative numbers (that is, complements of positive numbers), thus converting the subtraction to an addition. Since multiplication and division correspond to repeated addition and subtraction, respectively, it is possible to perform the four basic arithmetic operations using only the hardware for addition when negative numbers are represented in complement form. There are two popular complement number systems, (1) *radix complement* and (2) *diminished radix complement*.

### A.7.1 Radix Complement System

This system is called either 2s complement or 10s complement system, depending on whether the base of the number is 2 or 10. We will discuss

the 2s complement system. The 10s complement system also displays the same characteristics as the 2s complement system.

The radix complement of a number $(N)_r$ is defined as

$$[N]_r = r^n - (N)_r \tag{A.3}$$

where $[N]_r$ is the radix complement of $(N)_r$ and $n$ is the number of digits in $(N)_r$.

---

**Example A.33** The 2s complement of $(01010)_2$ is $2^5 - (01010)_2$; $n = 5$, $r = 2$.

$$= 100000 - 01010$$
$$= 10110$$
$$\therefore [01010]_2 = (10110)_2.$$

---

There are two other methods for obtaining the 2s complement of a number.

*Method 1*

$[01010]_2 = ?$

10101 ← Complement each bit (i.e., change each 0 to 1 and 1 to 0).

$\underline{+\ 1}$ ← Add 1 to the LSB to get the 2s complement.

10110

*Method 2*

$[010 \mid 10]_2 = ?$

10 ← Copy the bits from LSB until and including the first nonzero bit.

101 ←Complement the remaining bits through MSB to get the 2s complement.

101 | 10

## Twos Complement Arithmetic

In the 2s complement system, positive numbers remain in their normal form while the negative numbers are represented in the 2s complement form. Some examples using a 5-bit (4-magnitude, 1-sign) representation are presented.

**Example A.34**

| | | |
|---|---|---|
| +5 | 0 0101 | |
| −5 | 1 1011 | 2s complement of +5 |
| +4 | 0 0100 | |
| −4 | 1 1100 | |

Examples of arithmetic using these 5-bit 2s complement numbers are shown in the examples below:

**Example A.35**

| | Sign–magnitude | 2s complement | |
|---|---|---|---|
| 5 | 0, 0101 | 0 0101 | |
| −4 | 1, 0100 | 1 1100 | (2s complement of +4) |

CARRY from sign position 10 0001 SUM
$= +(0001)_2$

The CARRY from the MSB (sign bit in this case) is ignored. The result is positive because the MSB is 0.

**Example A.36**

| | Sign–magnitude | 2s complement |
|---|---|---|
| 4 | 0, 0100 | 0 0100 |
| −5 | 1, 0101 | 1 1011 |
| | | 1 1111 SUM |
| | | $= -(0001)_2$ |

There is no CARRY from the MSB. Since MSB is 1, the result is negative and must be 2s-complemented to find its value, that is, $-(0001)_2$ or $-1$.

To summarize, in 2s complement addition if the MSB generates a CARRY, it is ignored and the result is treated as positive; if there is no CARRY, the result is negative and is in the 2s complement format. Note that in this arithmetic the sign bit is treated as though it is one of the magnitude bits.

### Shifting

Shifting a 2s complement number left or right by one bit is equivalent to multiplying it by 2 or dividing it by 2, respectively. The sign bit is copied into the vacant position during a right shift and 0 is inserted into LSB during a left shift. An example follows.

---

**Example A.37**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Right shift (copy sign bit.) |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Right shift (copy sign bit.) |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Left shift (Insert 0.) |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Left shift (Insert 0.) |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Left shift (Insert 0) , OVERFLOW. |

Change in the value of the sign bit during a left shift indicates that there is an overflow (i.e., the result is too large), as shown by the last case above in this example.

---

## A.7.2 Diminished Radix Complement

The diminished radix complement $[N]_{r-1}$ of a number $(N)_r$ is defined as

$$[N]_{r-1} = r^n - (N)_r - 1 \tag{A.4}$$

where $n$ is the number of digits in $(N)_r$. Note that

$$[N]_r = [N]_{r-1} + 1. \tag{A.5}$$

The diminished radix complement is called 1s complement or 9s complement, depending on whether the base of the system is 2 or 10, respectively.

Since $(2^n - 1)$ is an $n$-bit binary number with all of its bits equal to 1, the 1s complement of a number is obtained by subtracting each of its bits from 1. This is the same as complementing each bit. As an example, the 1s complement of $(10110)_2$ is $(01001)_2$. The 9s complement of a decimal number is obtained by subtracting each of its digits from 9.

## Addition

In the 1s complement number system, the positive numbers remain in their true binary form while the negative numbers are represented in the 1s complement form. The following examples illustrate the arithmetic in this system using a 5-bit (4-magnitude, 1-sign) representation.

---

**Example A.38**

| | Sign–magnitude | 1s complement | |
|---|---|---|---|
| 5 | 0,0101 | 0 0101 | |
| −4 | 1,0100 | 1 1011 | |
| | | 1 0 0000 | SUM |
| Add the CARRY end-round → | | 1 | |
| | | 0 0001 | |

The CARRY generated from MSB during the addition is added (end-around) to the LSB to obtain the correct SUM; the result is positive., because MSB = 0.

---

**Example A.39**

| | Sign–magnitude | 1s complement |
|---|---|---|
| 4 | 0,0100 | 0 0100 |
| −5 | 1,0101 | 1 1010 |
| | | 1 1110 SUM |
| | | ↑ Negative. |
| | | ∴ Complement and the result is −(0001). |

---

## Shifting

While shifting 1s complement numbers right, the sign bit is copied into the vacant positions on the left (as in the 2s complement system); during a left shift, sign bit is copied into the vacant positions on the right.

Table A.6 summarizes the operations in both complement systems. Some digital systems use a 2s complement system while others employ 1s complement.

**Table A.6** Complement Number Systems

| Operation | Result | 1s complement | 2s complement |
|---|---|---|---|
| ADD | MSB is 1 | Result is in 1s complement | Result is in 2s complement |
| | CARRY from MSB | Add 1 to LSB of the result | Neglect the CARRY |
| Shift left | | Copy sign bit on the right | Copy 0s on the right |
| Shift right | | Copy sign bit on the left | Copy sign bit on the left |

## A.8 CODES

A digital system requires that all its information be in binary form. But the external world uses the alphabetic characters, decimal digits, and special characters (e.g., periods, commas, plus and minus signs) to represent information. A unique pattern of 0s and 1s is used to represent each required character. This pattern is the "code" corresponding to that character. We will list several codes that are commonly used in digital systems.

### A.8.1 Binary Coded Decimal (BCD)

BCD is one of the popular codes used to represent numeric information. Each decimal digit is coded into 4 bits:

| | | | |
|---|---|---|---|
| 0 | 0000 | 5 | 0101 |
| 1 | 0001 | 6 | 0110 |
| 2 | 0010 | 7 | 0111 |
| 3 | 0011 | 8 | 1000 |
| 4 | 0100 | 9 | 1001 |

Only ten of the possible sixteen ($2^4$) patterns of bits are used. As an example, the number $(432)_{10}$ would be represented as

$$\underset{4}{(0100} \quad \underset{3}{0011} \quad \underset{2}{0010)_{BCD}}$$

When the data are represented in BCD, the arithmetic can also be done on each digit, as in decimal arithmetic.

### A.8.2 Excess-3

Excess-3 is also a 4-bit code. The code for each decimal digit is obtained by adding 3 to the corresponding BCD code:

| | | | |
|---|---|---|---|
| 0 | 0011 | 5 | 1000 |
| 1 | 0100 | 6 | 1001 |
| 2 | 0101 | 7 | 1010 |
| 3 | 0110 | 8 | 1011 |
| 4 | 0111 | 9 | 1100 |

Excess-3 code enables simpler BCD arithmetic hardware. For example, the 9s complement of any digit can be obtained simply by taking the 1s complement of the corresponding code.

Consider the addition of two decimal digits. In BCD, when the SUM exceeds 9, a 6 is added to bring the resulting digit into the range 0–9. A CARRY is generated in this process that is used in the addition of digits in the next stage.

In Excess-3 it is not necessary to check to see if the SUM exceeds 9 or not since the correction can be done just by checking to see if the addition has resulted in a CARRY or not. The correction is to either add 3 or subtract 3 from the result, as shown by Example A.40.

---

**Example A.40**

| Decimal | BCD | Excess-3 |
|---|---|---|
| 3 | 0011 | 0110 |
| + 2 | 0010 | 0101 |
| 5 | 0101 | $\boxed{0}$ 1011 |
| | SUM < 9: No correction needed. | CARRY = 0<br>∴ Subtract 0011<br>3 1000 |
| 5 | 0101 | 1000 |
| + 6 | 0010 | 1001 |
| 11 | 1011 | 10011 |
| | SUM > 9:<br>∴ Add 6: 0110<br>1,0001 | CARRY = 1<br>∴ Add 3: 0011<br>1,0100 |

---

### A.8.3 Two-out-of-five Code

This code uses 5 bits to represent each decimal digit. As the name implies, 2 bits out of the 5 bits are 1s in each code word:

| | | | |
|---|---|---|---|
| 0 | 11000 | 5 | 01010 |
| 1 | 00011 | 6 | 01100 |
| 2 | 00101 | 7 | 10001 |
| 3 | 00110 | 8 | 10010 |
| 4 | 01001 | 9 | 10100 |

With 5 bits, 32 code words are possible. Only 10 of them are used, but the code provides an automatic error detection capability since the number of 1s in the code word must always be two.

This code uses a simple "parity" scheme. Since the number of 1s is always even, it is called "even parity." In general, one or more parity bits are included into the code word to facilitate error detection and correction. Even and odd parity schemes are the simplest ones. Several other schemes are used to generate parity bits where a higher degree of error detection and correction is needed.

### A.8.4 Alphanumeric Codes

When alphabetic characters, numeric digits, and special characters are used in representing the information to be processed by the digital system, an alphanumeric code is used. Two popular alphanumeric codes are Extended BCD Interchange Code (EBCDIC) and American Standard Code for Information Interchange (ASCII). These codes are shown in Table A.7.

**Table A.7** Alphanumeric Codes

| Character | EBCDIC code | ASCII code |
|---|---|---|
| blank | 0100 0000 | 0010 0000 |
| . | 0100 1011 | 0010 1110 |
| ( | 0100 1101 | 0010 1000 |
| + | 0100 1110 | 0010 1011 |
| $ | 0101 1011 | 0010 0100 |
| * | 0101 1100 | 0010 1010 |
| ) | 0101 1101 | 0010 1001 |
| – | 0110 0000 | 0010 1101 |

**Table A.7** Alphanumeric Codes (Continued)

| Character | EBCDIC code | ASCII code |
|---|---|---|
| / | 0110 0001 | 0010 1111 |
| , | 0110 1011 | 0010 1100 |
| ' | 0111 1101 | 0010 0111 |
| = | 0111 1110 | 0011 1101 |
| 0 | 1111 0000 | 0011 0000 |
| 1 | 1111 0001 | 0011 0001 |
| 2 | 1111 0010 | 0011 0010 |
| 3 | 1111 0011 | 0011 0011 |
| 4 | 1111 0100 | 0011 0100 |
| 5 | 1111 0101 | 0011 0101 |
| 6 | 1111 0110 | 0011 0110 |
| 7 | 1111 0111 | 0011 0111 |
| 8 | 1111 1000 | 0011 1000 |
| 9 | 1111 1001 | 0011 1001 |
| A | 1100 0001 | 0100 0001 |
| B | 1100 0010 | 0100 0010 |
| C | 1100 0011 | 0100 0011 |
| D | 1100 0100 | 0100 0100 |
| E | 1100 0101 | 0100 0101 |
| F | 1100 0110 | 0100 0110 |
| G | 1100 0111 | 0100 0111 |
| H | 1100 1000 | 0100 1000 |
| I | 1100 1001 | 0100 1001 |
| J | 1101 0001 | 0100 1010 |
| K | 1101 0010 | 0100 1011 |
| L | 1101 0011 | 0100 1100 |
| M | 1101 0100 | 0100 1101 |
| N | 1101 0101 | 0100 1110 |
| O | 1101 0110 | 0100 1111 |
| P | 1101 0111 | 0101 0000 |
| Q | 1101 1000 | 0101 0001 |
| R | 1101 1001 | 0101 0010 |
| S | 1110 0010 | 0101 0011 |
| T | 1110 0011 | 0101 0100 |
| U | 1110 0100 | 0101 0101 |
| V | 1110 0101 | 0101 0110 |
| W | 1110 0110 | 0101 0111 |
| X | 1110 0111 | 0101 1000 |
| Y | 1110 1000 | 0101 1001 |
| Z | 1110 1001 | 0101 1010 |

## REFERENCES

Kohavi, Z. *Switching and Automata Theory*, New York, NY: McGraw-Hill, 1978.
McCluskey, E. J. *Introduction to the Theory of Switching Circuits*, New York, NY: McGraw-Hill, 1965.
Mano, M. M. Digital Design, Englewood Cliffs, NJ: Prentice-hall, 1991.
Shiva, S. G. *Introduction to Logic Design*, New York, NY; Marcel Dekker, 1998.

# Appendix B
## Minimization of Boolean Functions

We used the theorems and postulates of Boolean algebra to simplify Boolean functions in Chapter 1. A minimized function yields a less complex circuit than a non-minimized function. In general, the complexity of a gate increases as the number of inputs increases. Hence, a reduction in the number of literals in a Boolean function reduces the complexity of the complete circuit. In designing integrated circuits (IC), there are other considerations, such as the area taken up by the circuit on the silicon wafer used to fabricate the IC and the regularity of the structure of the circuit from a fabrication point of view. For example, a programmable logic array (PLA) implementation (see Chapter 3) of the circuit yields a more regular structure than the random logic (i.e., using gates) implementation. Minimizing the number of literals in the function may not yield a less complex PLA implementation. However, if some product terms in the SOP form can be completely eliminated from the function, the PLA size can be reduced.

Minimization using theorems and postulates is tedious. Two other popular minimization methods are (1) using Karnaugh maps (K-maps), and (2) the Quine–McCluskey procedure. These two methods are described in this appendix.

### B.1 VENN DIAGRAMS

Truth tables and canonical forms were used in Chapter 1 to represent Boolean functions. Another method of representing a function is by using Venn diagrams. The variables are represented as circles in a "universe" that is a rectangle. The universe corresponds to 1 (everything), and 0 corresponds to null (nothing). Figures B.1 and B.2 show typical logic operations using Venn diagrams. In these diagrams, the NOT operation is identified by the area "NOT belonging" to the particular

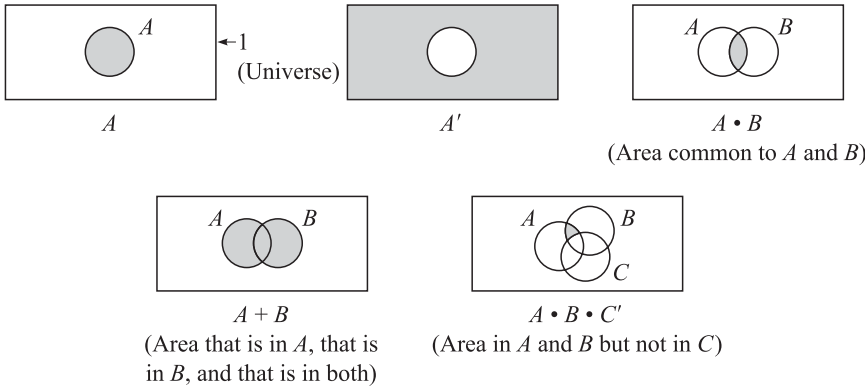



**Figure B.1** Logic operations using Venn diagrams

variable; the OR operation is the "union" of two areas (that is, the area that belongs to either or both) corresponding to the two operands; and the AND operation is the "intersection" of the two areas (that is, the area that is common to both) corresponding to the two operands. The unshaded area is the area in which the expression is 0. Note that all the combinations shown in the truth tables can also be shown in the Venn diagrams. Fig. B.3 shows all the combinations corresponding to two- and three-variable functions.

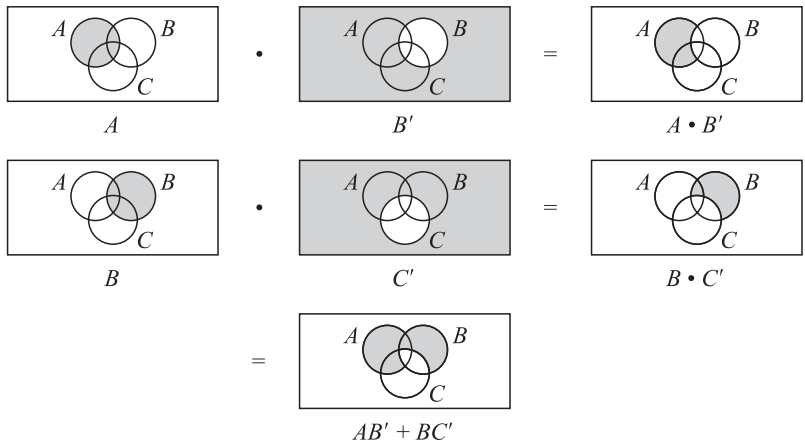



**Figure B.2** Representation of $AB' + BC'$ using Venn diagrams

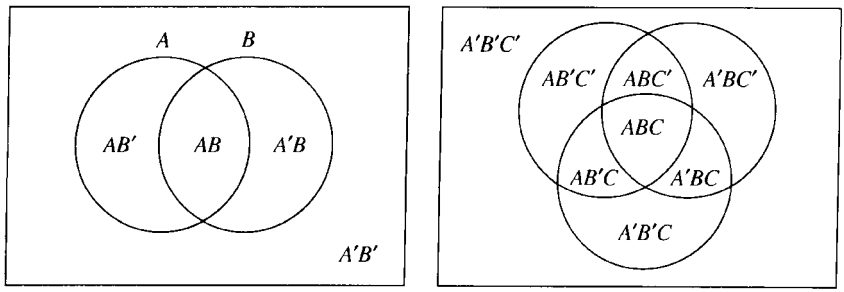
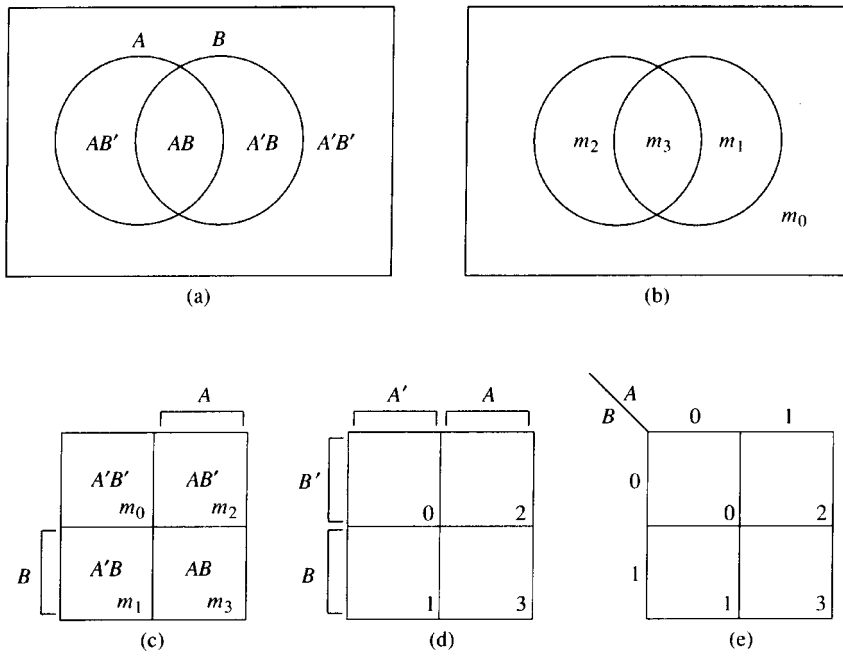



**Figure B.3** Venn diagram designating all possible combinations of variables

## B.2 KARNAUGH MAPS

Karnaugh maps (K-maps) are modified Venn diagrams. Consider the two-variable Venn diagram shown in Fig. B.4(a). All four combinations of the two variables are shown. The four areas are identified by the four minterms in (b), and (c) shows the Venn diagram rearranged such that the four areas are equal. Also note that the two right-hand blocks of the diagram correspond to $A$ ($m_2$ and $m_3$), and the two blocks at the bottom ($m_1$ and $m_3$) correspond to $B$. Figure B.4(d) marks the areas $A$, $A'$, $B$, and $B'$ explicitly, and (e) is the usual form for a K-map of two variables. The two variables $A$ and $B$ are distributed such that the values of $A$ are along the top and those of $B$ are along the side.

Figure B.5 shows a three-variable K-map. Since there are $2^3$ or 8 combinations of three variables, we will need eight blocks. The blocks are arranged such that the two right-hand columns correspond to $A$, the two middle columns correspond to $B$, and the bottom row corresponds to $C$. Each block corresponds to a minterm. For example, the block named $m_6$ corresponds to the area in $A$ and $B$ but not in $C$. That is the area $ABC'$, which is 110 in minterm code and is minterm $m_6$. The first two variables $A$ and $B$ are represented by the four combinations along the top and the third variable $C$ along the side as in Fig. B.5(b). Note that the area $A$ consists of the blocks where $A$ has a value of 1 (blocks 4, 5, 6, and 7), irrespective of $B$ and $C$; similarly, $B$ is 1 in blocks 2, 3, 6, and 7 and $C$ is 1 in 1, 3, 5, and 7. Once the variable values are listed along the top and side, it is very easy to identify the minterm corresponding to each block. For example, the left-hand, top-corner block corresponds to $A = 0$, $B = 0$, and $C = 0$; that is, $ABC = 000 = m_0$.




*Note*: $m_0$, $m_1$, $m_2$, and $m_1$ are minterms.

**Figure B.4** Two-variable Karnaugh map

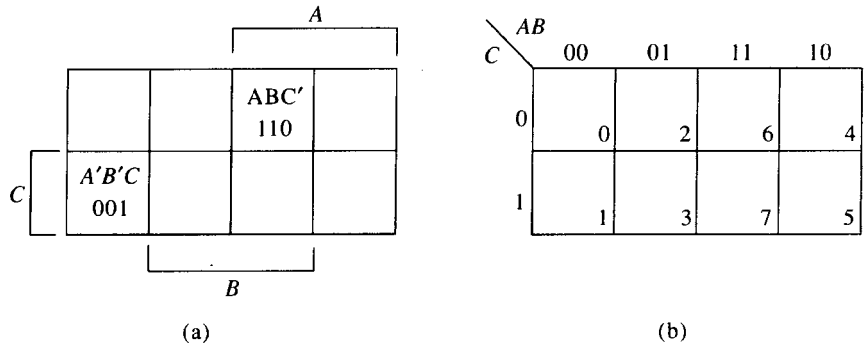
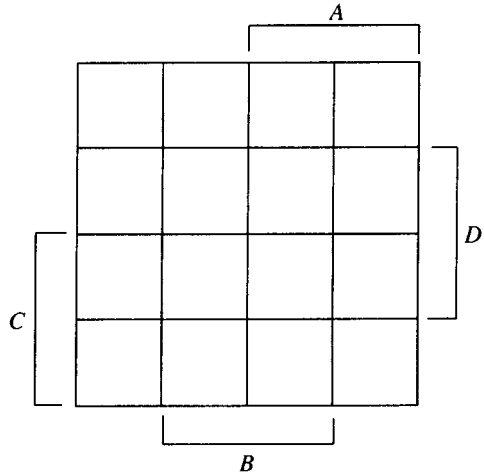



**Figure B.5** Three-variable Karnaugh map

A four-variable Karnaugh map is shown in Fig. B.6. The areas and minterms are also identified.

### B.2.1 Representation of Functions on K-maps

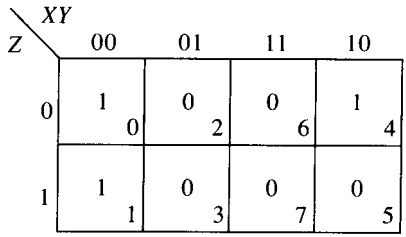
We represented functions on Venn diagrams by shading the areas. On Karnaugh maps, each block is given a value of a 0 or 1 depending on the value of the function. Each block corresponding to a minterm will have a value of 1; all other blocks will have 0s as shown in Examples B.1 and B.2.



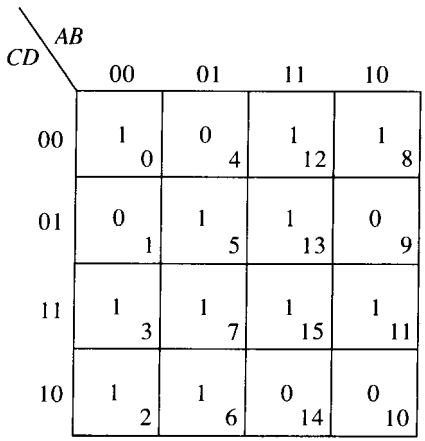

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 4 | 12 | 8 |
| 01 | 1 | 5 | 13 | 9 |
| 11 | 3 | 7 | 15 | 11 |
| 10 | 2 | 6 | 14 | 10 |

**Figure B.6** Four-variable Karnaugh map

**Example B.1** $f(X, Y, Z) = \Sigma m(0, 1, 4)$. Place a 1 corresponding to each minterm.




**Example B.2** $f(A, B, C, D) = \Pi_M(1, 4, 9, 10, 14)$. Place a 0 corresponding to each maxterm.

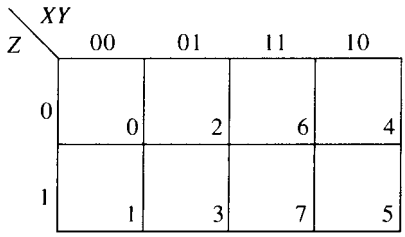



Usually 0s are not shown explicitly on the K-map. Only 1s are shown and a blank block corresponds to a 0.

### B.2.2 Plotting Sum of Products Form

When the function is given in the sum of products (SOP) form, the equivalent minterm list can be derived by the method given in Chapter 1 and the minterms can be plotted on the K-map. An alternative and faster method is to intersect the areas on the K-map corresponding to each product term as illustrated in Example B.3.

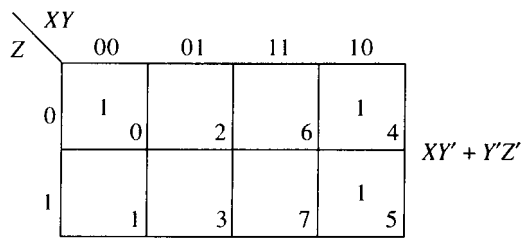
**Example B.3** $F(X, Y, Z) = XY' + Y'Z'$.




$X$ corresponds to blocks 4, 5, 6, 7 (all the blocks where $X$ is 1); $Y'$ corresponds to blocks 0, 1, 4, 5 (all the blocks where $Y$ is 0): $XY'$ corresponds to their intersection; that is, $XY' = \underline{\underline{4,5}}$. Similarly,

$$Y' = 0, 1, 4, 5$$
$$Z' = 0, 2, 4, 6 \qquad \therefore Y'Z' = \underline{\underline{0,4}}.$$

Therefore, the K-map will have 1 in the union of (4,5) and (0,4), which is (0,4,5):



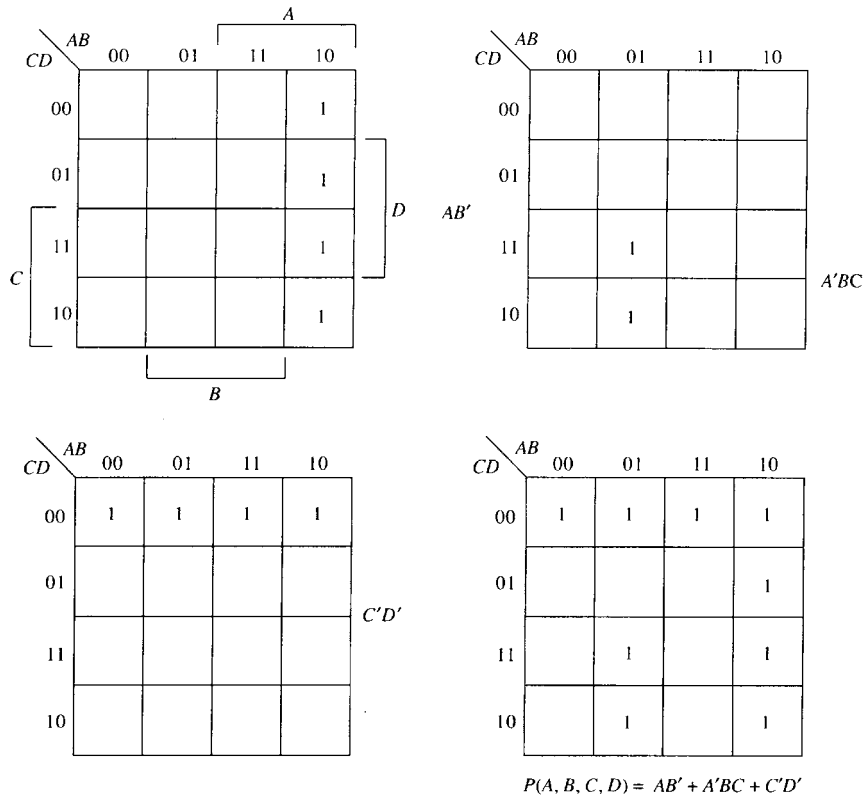

Alternatively, $XY'$ corresponds to the area where $X = 1$ *and* $Y = 0$, which is the last column; $Y'Z'$ corresponds to the area where both $Y = 0$ and $Z = 0$, which is blocks 0 and 4. Hence the union of the two corresponds to blocks 0,4,5.

Note also that in this three-variable K-map, if a product term has two variables missing (as in $Y$), we use four 1s corresponding to the four minterms that can be generated out of a single-variable product term in the representation.

In general, a product term with $n$ missing variables will be represented by $2^n$ 1s on the K-map. An example follows.

---

**Example B.4** $P(A, B, C, D) = AB' + A'BC + C'D'$.

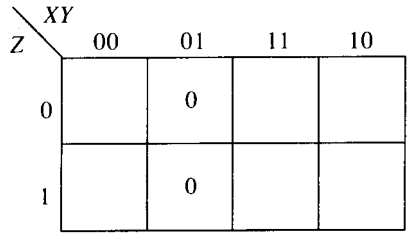



---

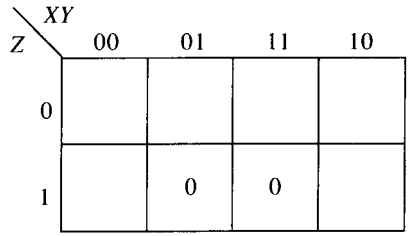
### B.2.3 Plotting Product of Sums Form

The procedure for plotting a product of sums (POS) expression is similar to that for the SOP form, except that 0s are used instead of 1s.

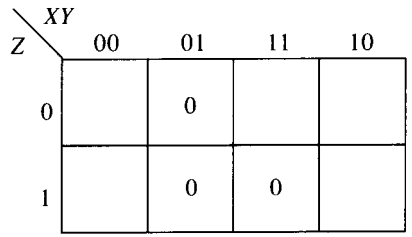
**Example B.5** $F(X, Y, Z) = (X + Y')(Y' + Z')$.



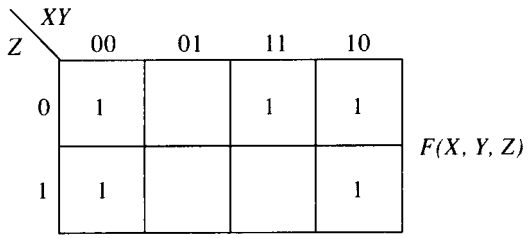

$(X + Y') = 0$ only if $X = 0$ and $Y' = 0$; that is, $X = 0$ and $Y = 1$ or the area $(X'Y)$.




$(Y' + Z') = 0$ only if $Y' = 0$ and $Z' = 0$; that is, $Y = 1$ and $Z = 1$ or the area $(YZ)$




$F(X, Y, Z)$ is 0 when either $(X + Y')$ is 0 or $(Y' + Z') = 0$ or the area $(X'Y) + (YZ)$.



### B.2.4 Minimization

Note that the combination of variable values represented by any block on a K-map differs from that of its adjacent block only in one variable, that variable being complemented in one block and true (or uncomplemented) in the other. For example, consider blocks 2 and 3 (corresponding to minterms $m_2$ and $m_3$) of a three-variable K-map: $m_2$ corresponds to 010 or $A'BC'$, and $m_3$ corresponds to 011 or $A'BC$. The values for $A$ and $B$ remain the same while $C$ is different in these adjacent blocks. This property where the two terms differ by only one variable is called *logical adjacency*. In a K-map, then, *physically adjacent blocks are also logically adjacent*. In the three-variable K-map, block 2 is physically adjacent to blocks 0, 3, and 6. Note that $m_2$ is also logically adjacent to $m_0$, $m_3$, and $m_6$. This adjacency property can be used in the simplification of Boolean functions.

---

**Example B.6** Consider the following K-map for a four-variable function:

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 4 | 1 12 | 1 8 |
| 01 | 1 | 1 5 | 1 13 | 1 9 |
| 11 | 3 | 7 | 15 | 11 |
| 10 | 2 | 6 | 14 | 10 |

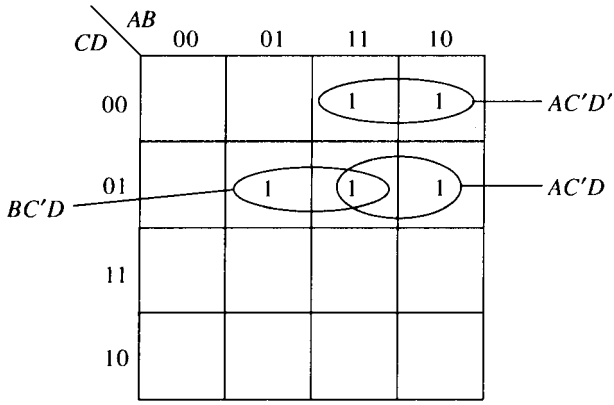
Blocks 8 and 12 are adjacent.

$$m_8 = 1000 = AB'C'D'.$$
$$m_{12} = 1100 = ABC'D'.$$

Also,

$$\begin{aligned} AB'C'D' + ABC'D' &= AC'D'(B' + B) && \text{P4b} \\ &= AC'D'.(1) && \text{P5a} \\ &= AC'D'. && \text{P1b} \end{aligned}$$

That is, we can combine $m_8$ and $m_{12}$. This combination is shown below by the grouping of 1s on the K-map. Note that by this grouping, we eliminated the variable $B$ because it changes in value between these two blocks.

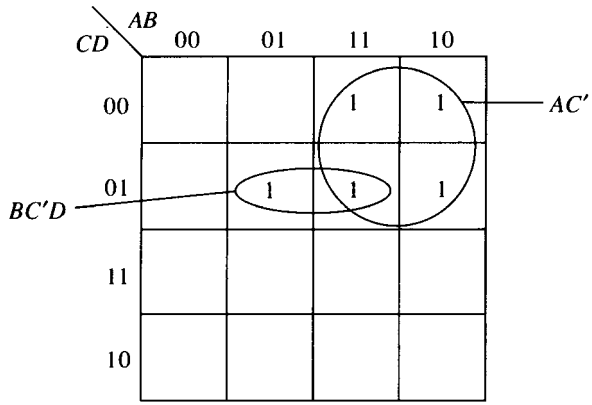



Similarly, the grouping of $m_9$ and $m_{13}$ yields $AC'D$ and grouping $m_5$ and $m_{13}$ yields $BC'D$.

If we combine $AC'D'$ with $AC'D$,

$$\begin{aligned} AC'D' + AC'D &= AC'(D + D') && \text{P4b} \\ &= AC' \cdot (1) && \text{P5a} \\ &= AC'. && \text{P1b} \end{aligned}$$

This in effect is equivalent to grouping all *four* 1s in the top right corner of the K-map, as shown here:



By forming a group of two adjacent 1s we eliminated 1 literal from the product term; by grouping four adjacent 1s we eliminated 2 literals. In general, if we group $2^n$ adjacent 1s, we can eliminate $n$ literals. Hence, in simplifying functions it is advantageous to form as large a group of 1s as possible. The number of 1s in any group must be a power of 2; that is, 1, 2, 4, 8,. . ., etc. Once the groups are formed, the product term corresponding to each group can be derived by the following general rules:

1. Eliminate the variable that changes in value within the group (move from block to block within the group to observe this change) from a product term containing all the variables of the function.
2. A variable that has a value of 0 in all blocks of the group should appear complemented in the product term.
3. A variable that has a value of 1 in all blocks of the group should appear uncomplemented in the product term.

For the group of four 1s in the above K-map:

$ABCD$ Start with all the variables.
$ABCD$ $A$ remains 1 in all four blocks.
$A\not{B}CD$ $B$ changes in value.
$A\not{B}C'D$ $C$ remains at 0.
$A\not{B}C'\not{D}$ $D$ changes in value.

So the product term corresponding to this grouping is $AC'$.

---

We can summarize all of the above observations in the following procedure for simplifying functions:

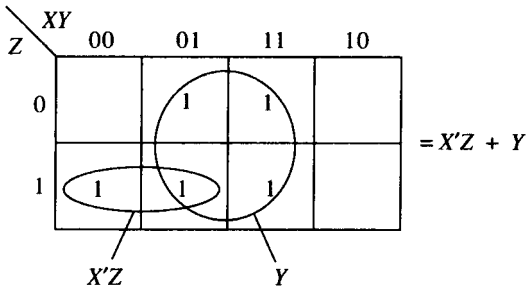
1. Form groups of adjacent 1s.
2. Form each group to be as large as possible. (The number of 1s in each group must be a power of 2.)
3. Cover each 1 on the K-map at least once. Same 1 can be included in several groups if necessary.
4. Select the least number of groups so as to cover all the 1s on the map.
5. Translate each group into a product term.
6. OR the product terms, to obtain the minimized function.

To recognize the adjacencies, on a three-variable map the right-hand edge is considered to be the same as the left-hand edge, thus making block 0 adjacent to block 4, and 1 adjacent to 5. Similarly, on a four-variable map, the top and bottom edges can be brought together to form a cylinder. The

two ends of the cylinder are brought together to form a toroid (like a donut). The following examples illustrate the grouping on the K-maps and corresponding simplifications.

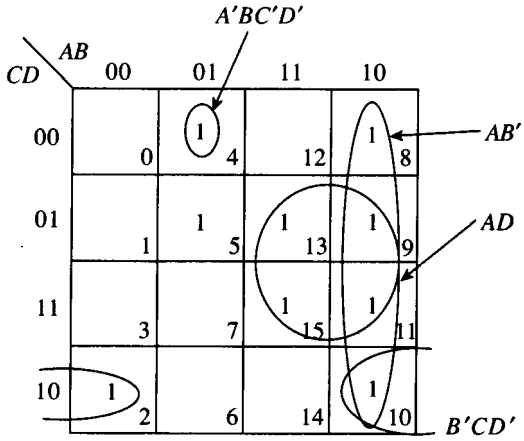
---

**Example B.7** $F(X, Y, Z) = \Sigma m(1, 2, 3, 6, 7)$.




---

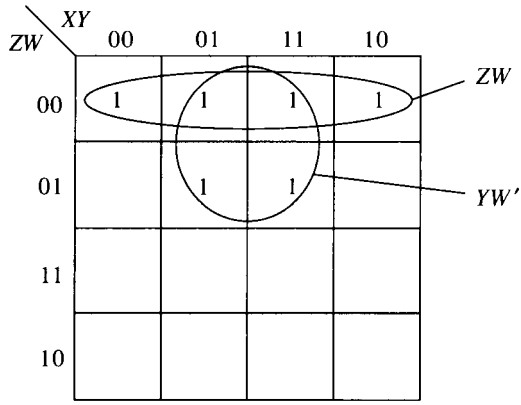
**Example B.8** $F(A, B, C, D) = \Sigma m(2, 4, 8, 9, 10, 11, 13, 15)$.




$F(A, B, C, D) = AB' + AD + B'CD' + A'BC'D'$.

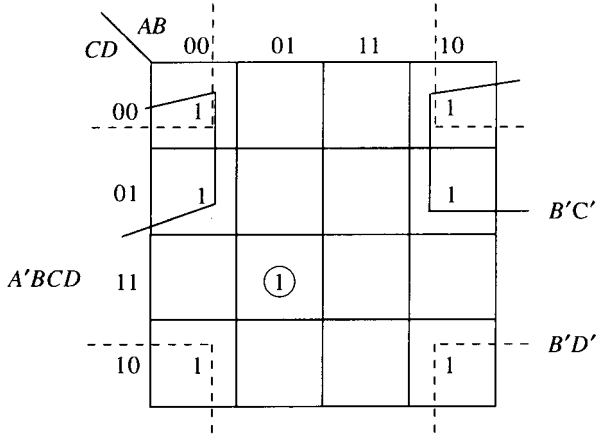
---

**Example B.9** $F(X, Y, Z, W) = \Sigma m(0, 4, 5, 8, 12, 13)$.



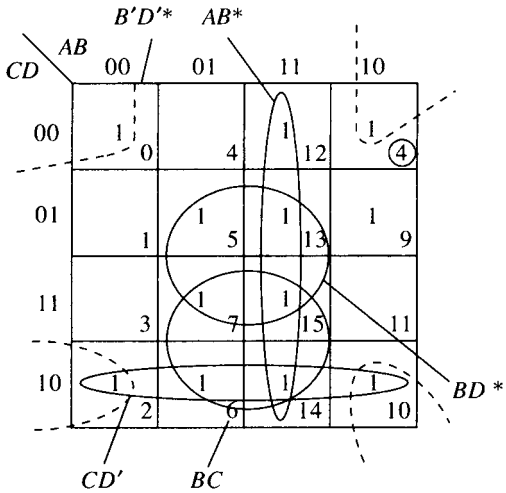

$F(X, Y, Z, W) = YW' + ZW$.

**Example B.10** $F(A, B, C, D) = \Sigma m(0, 1, 2, 7, 8, 9, 10)$.




$F(A, B, C, D) = A'BCD + B'D' + B'C'$.

**Example B.11** $F(A, B, C, D) = ABC' + ABC + BCD' + BCD + AB'D' + A'B'D' + A'BC'D$.




1. Groupings marked by an "*" are "essential." $m_{12}$ is covered only by $(AB)$: $m_0$, $m_8$ are covered only by $(B'D')$; $m_5$ is covered only by $(BD)$.
2. Once the above three groups are chosen, the only minterms left uncovered are $m_6$ and $m_{14}$. To cover these, we can either choose $(BC)$ or $(CD')$. Hence, there are two simplified forms:

$$F(A, B, C, D) = AB + B'D' + BD + BC$$
$$= AB + B'D' + BD + CD'.$$

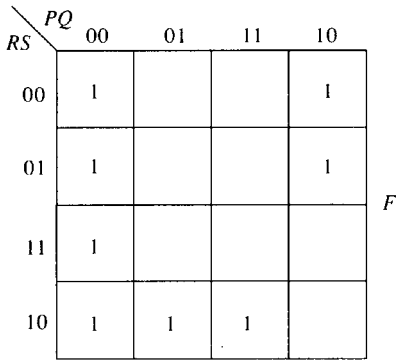
Either of the above is a satisfactory form, since each contains the same number of literals.

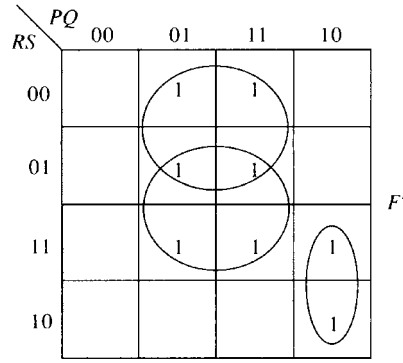
### B.2.5 Simplified Function in POS Form

To obtain the simplified function in POS form,

1. Plot the function $F$ on the K-map.
2. Derive the K-map for $F'$ (by changing 1 to 0 and 0 to 1).
3. Simplify the K-map for $F'$ to obtain $F'$ in SOP form.
4. Use DeMorgan's laws to obtain $F$.

**Example B.12** $F(P, Q, R, S) = P'Q'R' + P'Q'RS + P'RS' + PQRS' + PQ'R'$.




$$F' = QR' + QS + PQ'R$$

$$F = F'' = (QR' + QS + PQ'R)'$$

$$= (QR')' \cdot (QS)' \cdot (PQ'R)' \quad \text{T5a}$$

$$= (Q' + R)(Q' + S')(P' + Q + R') \quad \text{T5b}$$

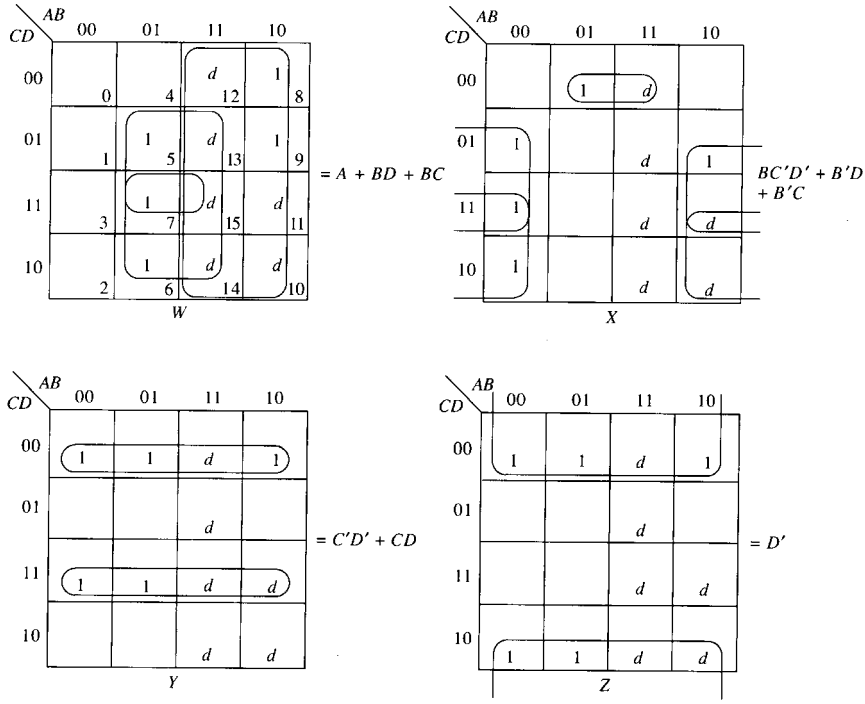
### B.2.6 Minimization Using Don't Cares

*Don't cares* are indicated by a "d" on the K-map. Each can be treated as either a 1 or a 0. It is not necessary to cover all the don't cares while grouping: that is, don't cares not covered are treated as 0s.

**Example B.13** The BCD-to-excess-3 decoder discussed in Chapter 1 expects as inputs only the combinations corresponding to decimals 0 through 9. The other six inputs will never occur. Hence, the output corresponding to each of these six inputs is a don't care.

The maps at the top of the next page illustrate the use of don't cares in simplifying the output functions of the decoder (refer to the truth table in Chapter 1).

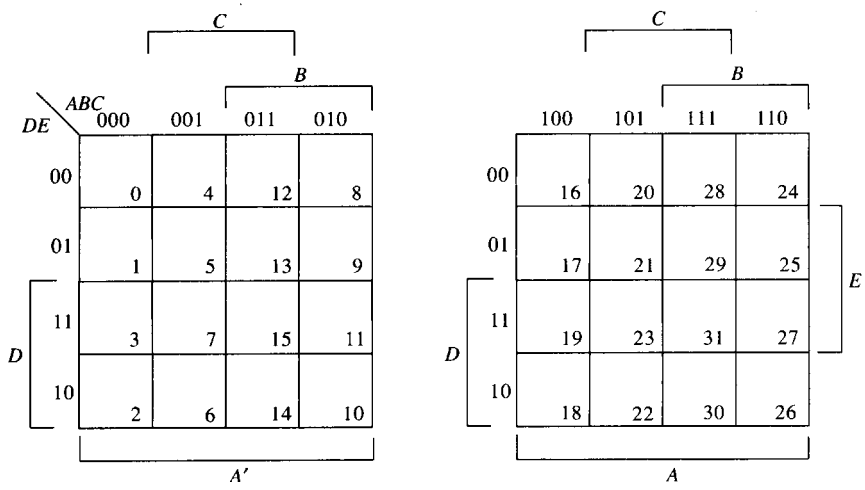
K-maps are useful for functions with up to four or five variables. Figure B.7 shows a five-variable K-map. Since the number of blocks doubles for each additional variable, minimization using K-maps becomes complex for functions with more than five variables. The Quine–McCluskey procedure described in the next section is useful in such cases.




## B.3 THE QUINE–McCLUSKEY PROCEDURE

The Quine–McCluskey procedure also uses the logical adjacency property to reduce the Boolean function. Two minterms are logically adjacent if they differ in one position only; that is, if one of the variables is in uncomplemented form in one minterm and complemented form in the other. Such a variable is eliminated by combining the two minterms. The Quine–McCluskey procedure compares each minterm with all the others and combines them if possible. The procedure uses the following steps:

1. Classify the minterms (and don't cares) of the function into groups such that each term in a group contains the same number of 1s in the binary representation of the term.
2. Arrange the groups formed in step 1 in the increasing order of number of 1s. Let the number of groups be $n$.




*Note*: The two parts of the map are treated as two planes, one superimposed over the other. The blocks at the same position on each plane are also logically adjacent.

**Figure B.7** Five-variable Karnaugh map

3. Compare each minterm in group $i (i = 1 \text{ to } n - 1)$ with those in group $(i + 1)$; if the two terms are adjacent, form a combined term. The variable thus eliminated is represented as "–" in the combined term.
4. Repeat the matching operation of step 3 on the combined terms until no more combinations can be done. Each combined term in the final list is called a *prime implicant* (PI). A prime implicant is a product term that cannot be combined with others to yield a term with fewer literals.
5. Construct a *prime implicant chart* in which there is one column for each minterm (only minterms; don't cares are not listed) and one row for each PI. An "*X*" in a row-column intersection indicates that the prime implicant corresponding to the row covers the minterm corresponding to the column.
6. Find all the essential prime implicants (i.e., the prime implicants that each cover at least one minterm that is not covered by any other PI).
7. Select a minimum number of prime implicants from the remaining, to cover those minterms not covered by the essential PIs.
8. The set of prime implicants thus selected form the minimum function.

This procedure is illustrated by Example B.14.

---

**Example B.14**

$F(A, B, C, D) = \Sigma m \underbrace{(0, 2, 4, 5, 6, 9, 10)}_{\text{Minterms}} + \Sigma_d \underbrace{(7, 11, 12, 13, 14, 15)}_{\text{Don't cares}}$.

*Steps 1 and 2*

| | | |
|---|---|---|
| $\surd$ 0 | 0000 | Group 0: Terms with no 1s. |
| $\surd$ 2 | 0010 | |
| $\surd$ 4 | 0100 | Group 1: Terms with 1. |
| $\surd$ 5 | 0101 | |
| $\surd$ 6 | 0110 | |
| $\surd$ 9 | 1001 | Group 2: Terms with two 1s. |
| $\surd$ 10 | 1010 | |
| $\surd$ 12 | 1100 | |
| $\surd$ 7 | 0111 | |
| $\surd$ 11 | 1011 | |
| $\surd$ 13 | 1101 | Group 3: Terms with three 1s. |
| $\surd$ 14 | 1110 | |
| $\surd$ 15 | 1111 | Group 4: Terms with four 1s. |

The "$\surd$" indicates that the term is used in forming a combined term at least once.

*Step 3*

| | | |
|---|---|---|
| $\surd$ (0, 2) | 00–0 | Obtained by matching groups 0 and 1. |
| $\surd$ (0, 4) | 0–00 | |
| $\surd$ (2, 6) | 0–10 | |
| $\surd$ (2, 10) | –010 | |
| $\surd$ (4, 5) | 010– | Obtained by matching groups 1 and 2. |
| $\surd$ (4, 6) | 01–0 | |
| $\surd$ (4, 12) | –100 | |
| $\surd$ (5, 7) | 01–1 | |
| $\surd$ (5, 13) | –101 | |
| $\surd$ (6, 7) | 011– | Obtained by matching groups 2 and 3. |
| $\surd$ (6,14) | –110 | |
| $\surd$ (9, 11) | 10–1 | |
| $\surd$ (9, 13) | 1–01 | |
| $\surd$ (10, 11) | 101– | |
| $\surd$ (10, 14) | 1–10 | |
| $\surd$ (12, 13) | 110– | |

| | | |
|---|---|---|
| √ (12,14 ) | 11–0 | |
| √ (7, 15) | –111 | |
| √ (11, 15) | 1–11 | Obtained by matching groups 3 and 4. |
| √ (13, 15) | 11–1 | |
| √ (14, 15) | 111– | |

*Step 4*

| | | |
|---|---|---|
| (0, 2, 4, 6) | 0– – 0 | Same (0, 4, 2, 6). |
| (2, 6, 10, 14) | – – 10 | Same as (2, 10, 6, 14). |
| √ (4, 5, 6, 7) | 01 – – | Same as (4, 6, 5, 7). |
| √ (4, 5, 12, 13) | – 10 – | Same as (4, 12, 5, 13). |
| √ (4, 6, 12, 14) | – 1–0 | Same as (4, 12, 6, 14). |
| √(5, 7, 13, 15) | – 1–1 | Same as (5, 13, 7, 15). |
| √(6, 7, 14, 15) | – 11 – | Same as (6, 14, 7, 15). |
| (9, 11, 13, 15) | 1– –1 | Same as (9, 13, 11, 15). |
| (10, 11, 14, 15) | 1–1 – | Same as (10, 14, 11, 15). |
| √(12, 13, 14, 15) | 11– – | Same as (12, 14, 11, 15). |

| | | | |
|---|---|---|---|
| (0, 2, 4, 6) | 0– –0 | $PI_1$ | |
| (2, 6, 10, 14) | – – 10 | $PI_2$ | |
| (4, 5, 6, 7, 12, 13, 14, 15) | – 1– – | $PI_3$ | Prime implicants. |
| (9, 11, 13, 15) | 1 – – 1 | $PI_4$ | |
| (10, 11, 14, 15) | 1 – 1 – | $PI_5$ | |

No further reduction possible.

*Step 5* Prime implicant chart

| | Minterms | | | | | | |
|---|---|---|---|---|---|---|---|
| | √ 0 | √ 2 | √ 4 | √ 5 | √ 6 | √ 9 | √ 10 |
| $PI_1$ | ⊗ | × | × | | × | | |
| $PI_2$ | | × | | | × | | × |
| $PI_3$ | | | × | ⊗ | × | | |
| $PI_4$ | | | | | | ⊗ | |
| $PI_5$ | | | | | | | × |

*Step 6* $PI_1$, $PI_3$, and $PI_4$ are "essential" since minterms 0, 5, and 9, respectively, are covered by only these PIs. These PIs together also cover minterms 2, 4, and 6.

*Step 7* To cover the remaining minterm 10, we can select either $PI_2$ or $PI_5$.

*Step 8* The reduced function is

$$\begin{aligned} F(A, B, C, D) &= PI_1 + PI_3 + PI_4 + PI_2 \text{ or } PI_5 \\ &= 0--0 + -1--+1--1+--10 \text{ or } 1-1- \\ &= (A'D' + B + AD + CD') \text{ or } (A'D' + B + AD + AC). \end{aligned}$$

---

The Quine–McCluskey procedure can be programmed on a computer and is efficient for functions of any number of variables.

## B.4 CONCLUSIONS

Several other techniques to simplify Boolean function have been devised. The interested reader is referred to the books listed under references. The automation of Boolean function minimization was an active research area in the seventies. The advent of LSI and VLSI has contributed to a decline of interest in the minimization of Boolean functions. The minimization of the number of ICs and the efficient interconnections between them is of more significance than the saving of a few gates in the present-day design environment.

## REFERENCES

Kohavi, Z. *Switching and Automata Theory*, New York, NY: McGraw-Hill, 1978.

McCluskey, E. J. *Introduction to the Theory of Switching Circuits*, New York, NY: McGraw-Hill, 1965.

Mano, M. M. *Digital Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

Shiva, S. G. *Introduction to Logical Design*, New York: Marcel Dekker, 1998.

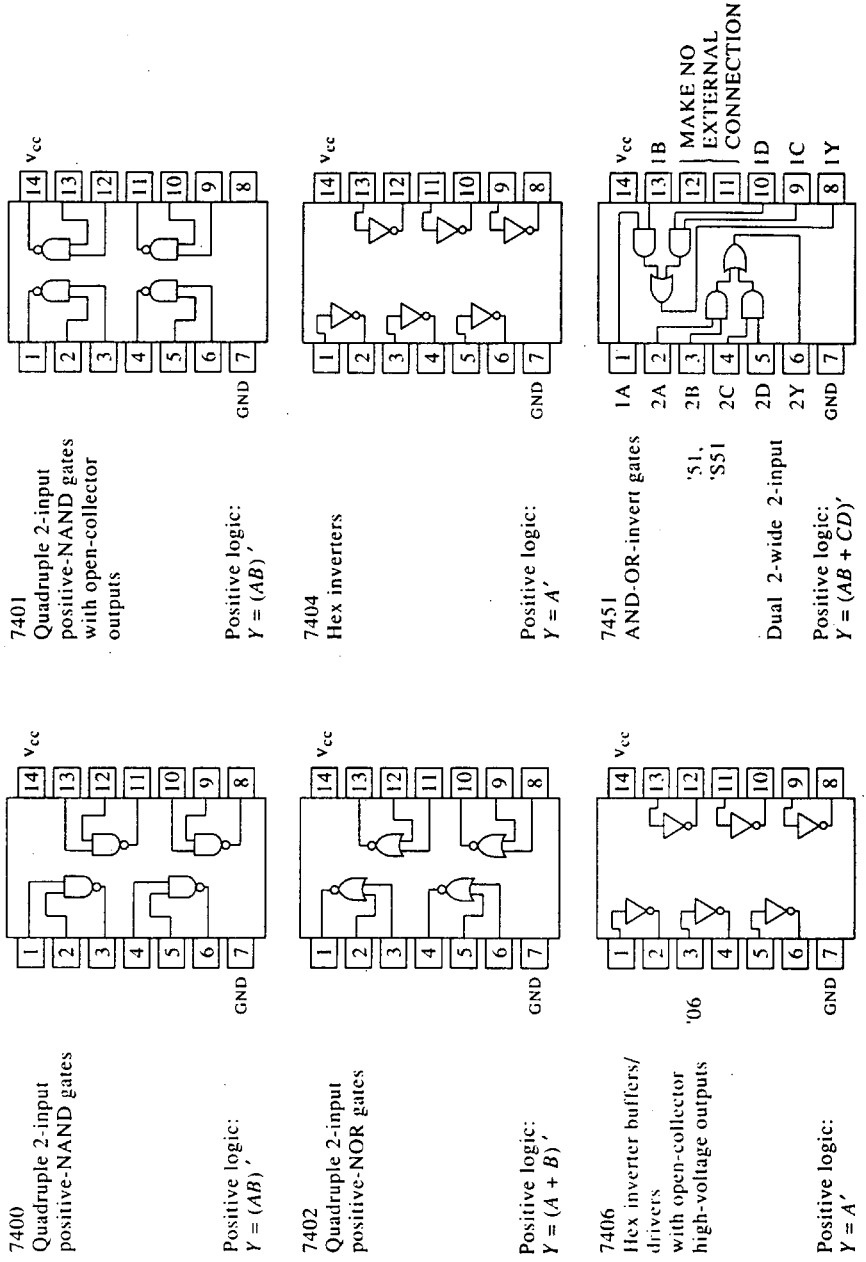
# Appendix C
## Details of Representative Integrated Circuits

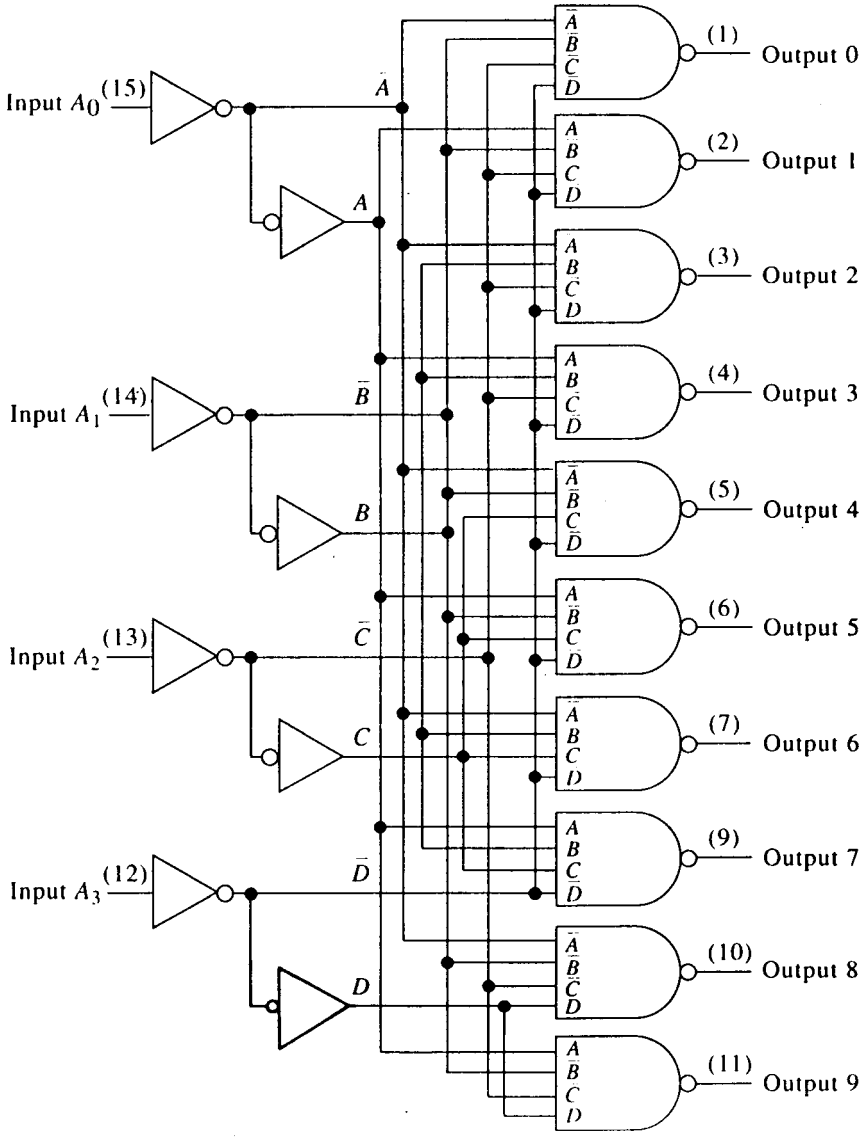
Pertinent details of representative integrated circuits (ICs) taken from vendor's manuals are reprinted in this Appendix. Refer to vendor's manuals for further details on these and other ICs. The ICs detailed here are grouped under:

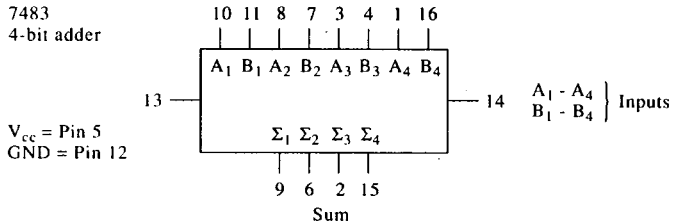
1. Gates, decoders and others ICs useful in combinational circuit design,
2. Flip-flops, registers and other ICs useful in sequential circuit design, and
3. Memory ICs.

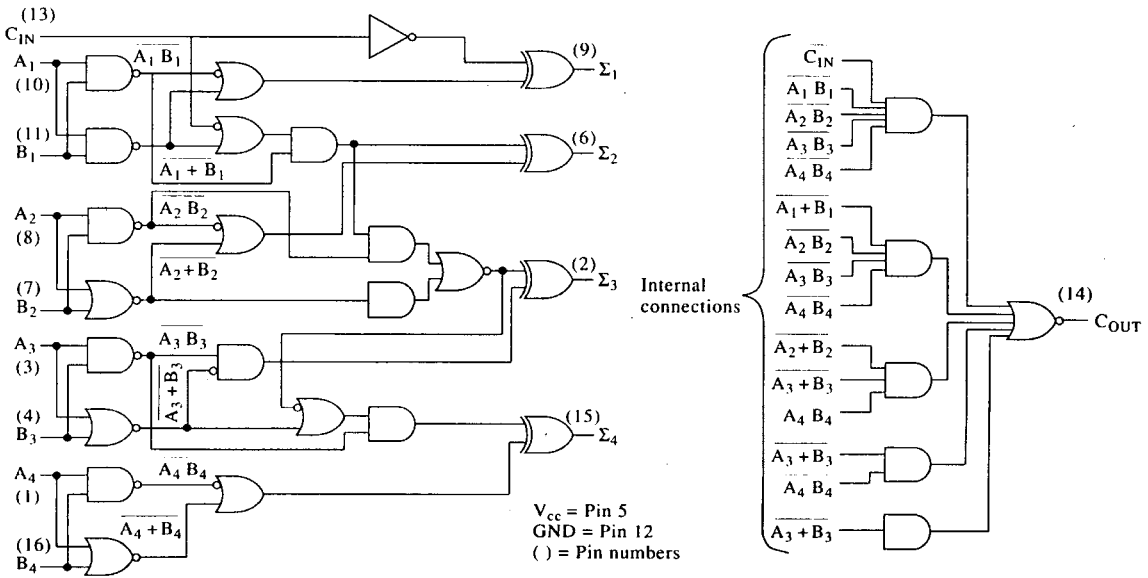
Most of the details provided here are from the TTL technology. No attempt is made to provide the details of the most up-to-date ICs. Because of the rapid changes in IC technology, it is hard to provide the latest details in a book. Refer to the current manuals from IC vendors for the latest information. Some of the ICs detailed here may no longer be available. Usually, alternative forms of these ICs or a later version in another technology can be found from the same vendor or from another source. Nevertheless, the details given here are representative of the characteristics a designer would seek.

## C.1 GATES, DECODERS AND OTHER ICs USEFUL IN COMBINATIONAL CIRCUIT DESIGN




Input $A_0$ (15)
Input $A_1$ (14)
Input $A_2$ (13)
Input $A_3$ (12)
$\bar{A}$
$A$
$\bar{B}$
$B$
$\bar{C}$
$C$
$\bar{D}$
$D$
(1) Output 0
(2) Output 1
(3) Output 2
(4) Output 3
(5) Output 4
(6) Output 5
(7) Output 6
(9) Output 7
(10) Output 8
(11) Output 9



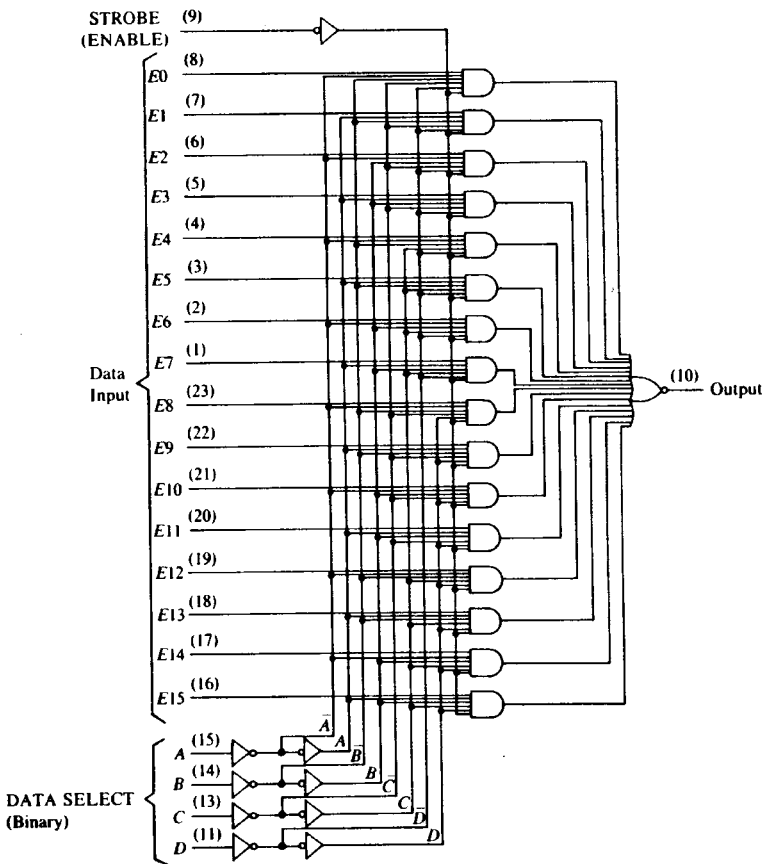




Note: " ¯ " is used to represent the NOT operation in IC catalogs.

Logic '150

Function table

| Inputs | | | | | Output |
|---|---|---|---|---|---|
| SELECT | | | | STROBE | |
| *D* | *C* | *B* | *A* | *S* | *W* |
| X | X | X | X | H | H |
| L | L | L | L | L | $\overline{E0}$ |
| L | L | L | H | L | $\overline{E1}$ |
| L | L | H | L | L | $\overline{E2}$ |
| L | L | H | H | L | $\overline{E3}$ |
| L | H | L | L | L | $\overline{E4}$ |
| L | H | L | H | L | $\overline{E5}$ |
| L | H | H | L | L | $\overline{E6}$ |
| L | H | H | H | L | $\overline{E7}$ |
| H | L | L | L | L | $\overline{E8}$ |
| H | L | L | H | L | $\overline{E9}$ |
| H | L | H | L | L | $\overline{E10}$ |
| H | L | H | H | L | $\overline{E11}$ |
| H | H | L | L | L | $\overline{E12}$ |
| H | H | L | H | L | $\overline{E13}$ |
| H | H | H | L | L | $\overline{E14}$ |
| H | H | H | H | L | $\overline{E15}$ |




74150 Data Selector/Multiplexer

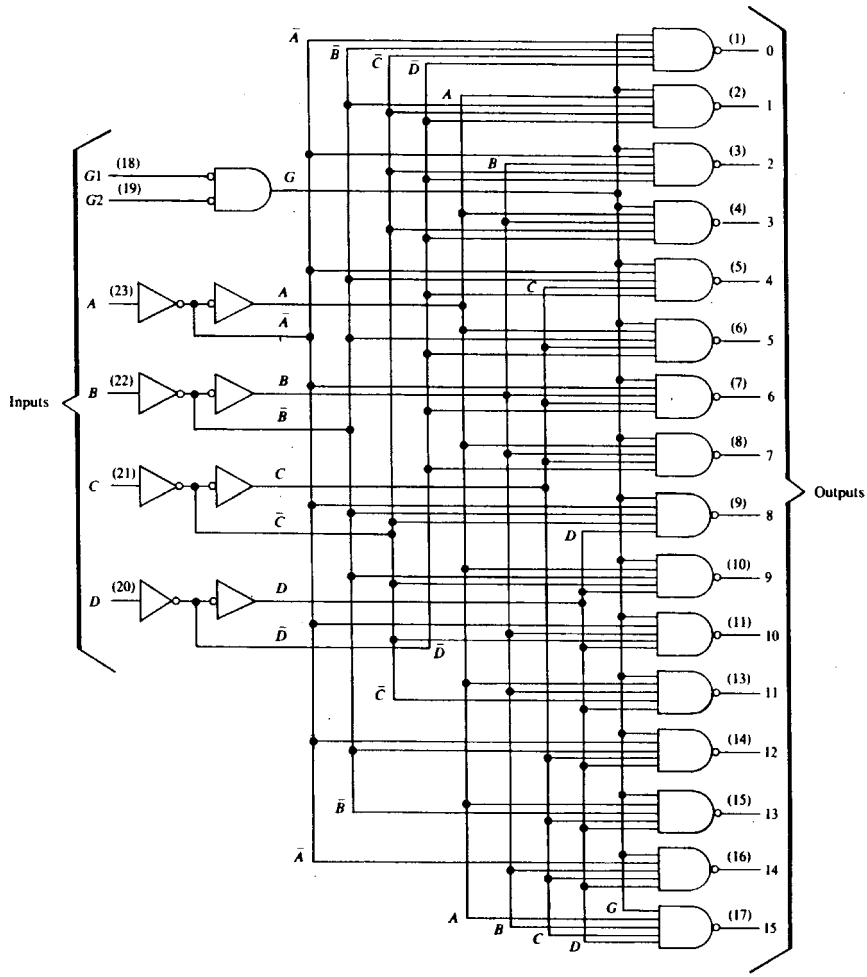
Logic

Function table

| Inputs | | | | | | Outputs | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *G1* | *G2* | *D* | *C* | *B* | *A* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| L | L | L | L | L | L | L | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H |
| L | L | L | L | L | H | H | L | H | H | H | H | H | H | H | H | H | H | H | H | H | H |
| L | L | L | L | H | L | H | H | L | H | H | H | H | H | H | H | H | H | H | H | H | H |
| L | L | L | L | H | H | H | H | H | L | H | H | H | H | H | H | H | H | H | H | H | H |
| L | L | L | H | L | L | H | H | H | H | L | H | H | H | H | H | H | H | H | H | H | H |
| L | L | L | H | L | H | H | H | H | H | H | L | H | H | H | H | H | H | H | H | H | H |
| L | L | L | H | H | L | H | H | H | H | H | H | L | H | H | H | H | H | H | H | H | H |
| L | L | L | H | H | H | H | H | H | H | H | H | H | L | H | H | H | H | H | H | H | H |
| L | L | H | L | L | L | H | H | H | H | H | H | H | H | L | H | H | H | H | H | H | H |
| L | L | H | L | L | H | H | H | H | H | H | H | H | H | H | L | H | H | H | H | H | H |
| L | L | H | L | H | L | H | H | H | H | H | H | H | H | H | H | L | H | H | H | H | H |
| L | L | H | L | H | H | H | H | H | H | H | H | H | H | H | H | H | L | H | H | H | H |
| L | L | H | H | L | L | H | H | H | H | H | H | H | H | H | H | H | H | L | H | H | H |
| L | L | H | H | L | H | H | H | H | H | H | H | H | H | H | H | H | H | H | L | H | H |
| L | L | H | H | H | L | H | H | H | H | H | H | H | H | H | H | H | H | H | H | L | H |
| L | L | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | L |
| L | H | X | X | X | X | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H |
| H | L | X | X | X | X | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H |
| H | H | X | X | X | X | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H | H |

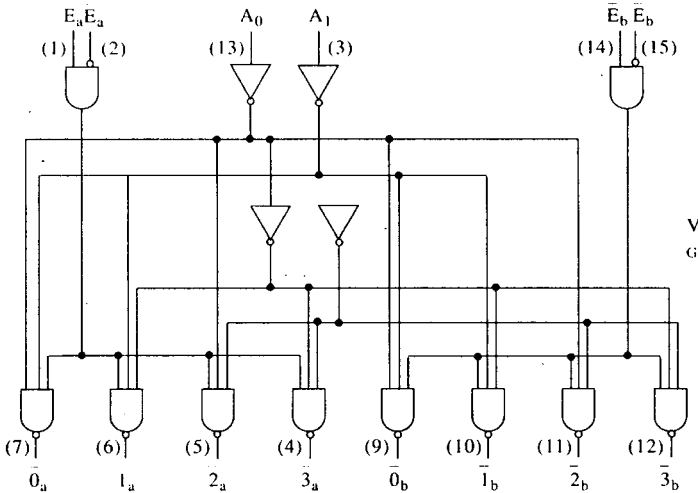
H = high level, L = low level, X = irrelevant

74154 4-line-to-16-line Decoder/Demultiplexer

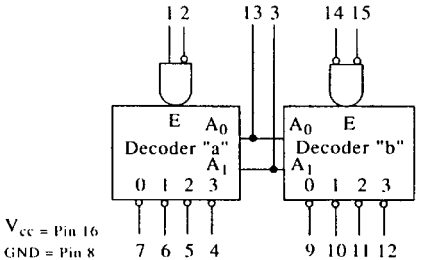



74154 (Continued)

LOGIC DIAGRAM




LOGIC SYMBOL




FUNCTION TABLE

| ADDRESS | | ENABLE "a" | | OUTPUT "a" | | | | ENABLE "b" | | OUTPUT "b" | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_0$ | $A_1$ | $E_a$ | $\overline{E}_a$ | $\overline{0}$ | $\overline{1}$ | $\overline{2}$ | $\overline{3}$ | $\overline{E}_b$ | $\overline{E}_b$ | $\overline{0}$ | $\overline{1}$ | $\overline{2}$ | $\overline{3}$ |
| X | X | L | X | H | H | H | H | H | X | H | H | H | H |
| X | X | X | H | H | H | H | H | X | H | H | H | H | H |
| L | L | H | L | L | H | H | H | L | L | L | H | H | H |
| H | L | H | L | H | L | H | H | L | L | H | L | H | H |
| L | H | H | L | H | H | L | H | L | L | H | H | L | H |
| H | H | H | L | H | H | H | L | L | L | H | H | H | L |

H = HIGH voltage level  L = LOW voltage level  X = Don't care

74155 Dual 2-line-to-4-line Decoder/Demultiplexer

## 74155

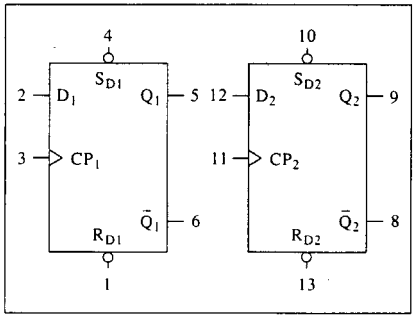
The '155 is a Dual 1-of-4 decoder–demultiplexer with common Address inputs and separate gated Enable inputs. Each decoder section, when enabled, will accept the binary weighted Address input ($A_0$, $A_1$) and provide four mutually exclusive active-LOW outputs ($\bar{0} - \bar{3}$). When the enable requirements of each decoder are not met, all outputs of that decoder are HIGH.

Both decoder sections have a 2-input enable gate. For decoder "a" the enable gate requires one active-HIGH input and one active-LOW input ($E_a \cdot \bar{E}_a$). Decoder "a" can accept either true or complemented data in demultiplexing applications, by using the $\bar{E}_a$ or $E_a$ inputs respectively. The decoder "b" enable gate requires two active-LOW inputs ($\bar{E}_b \cdot \bar{E}_b$). The device can be used as a 1-of-8 decoder/demultiplexer by tying $E_a$ to $\bar{E}_b$ and relabeling the common connection address as ($A_2$); forming the common enable by connecting the remaining $\bar{E}_b$ and $\bar{E}_a$.

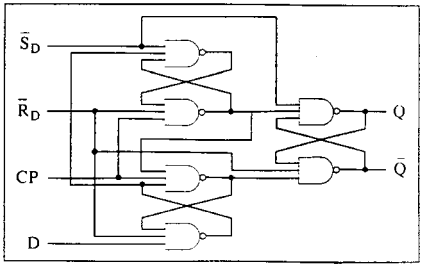
## C.2 FLIP-FLOPS, REGISTERS AND OTHER ICs USEFUL IN SEQUENTIAL CIRCUIT DESIGN

7474 *D* flip-flop IC

LOGIC SYMBOL




LOGIC DIAGRAM




Note: "−" indicates the NOT operation.

AC SET-UP REQUIREMENTS $T_A$=25°C, $V_{CC}$ = 5.0 V

| PARAMETER | TEST CONDITIONS | 74 | |
|---|---|---|---|
| | | Min | Max |
| $t_W$(H) Clock pulse width (HIGH) | Waveform 1 | 30 | |
| $t_W$(L) Clock pulse width (LOW) | Waveform 1 | 37 | |
| $t_W$(L) Set or reset pulse width (LOW) | Waveform 2 | 30 | |
| $t_s$(H) Set-up time (HIGH) data to clock | Waveform 1 | 20 | |
| $t_s$(L) Set-up time (LOW) data to clock | Waveform 1 | 20 | |
| $t_h$ Hold time data to clock | Waveform 1 | 5 | |

MODE SELECT — FUNCTION TABLE

| OPERATING MODE | INPUTS | | | | OUTPUTS | |
|---|---|---|---|---|---|---|
| | $\bar{S}_D$ | $\bar{R}_D$ | CP | D | Q | $\bar{Q}$ |
| Asynchronous Set | L | H | X | X | H | L |
| Asynchronous Reset (Clear) | H | L | X | X | L | H |
| Undetermined(1) | L | L | X | X | H | H |
| Load "1" (Set) | H | H | ↑ | h | H | L |
| Load "0" (Reset) | H | H | ↑ | l | L | H |

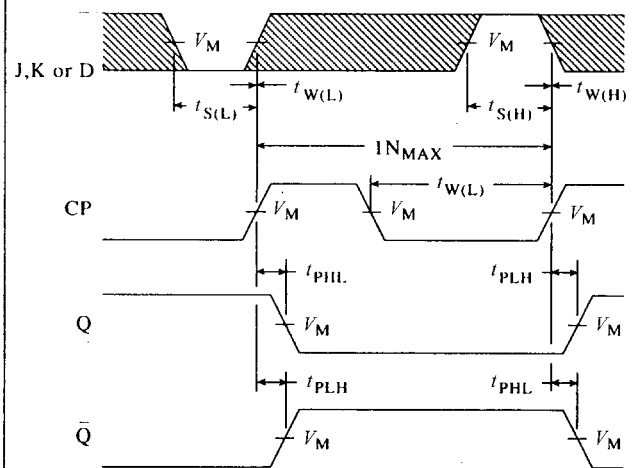
H = HIGH voltage level steady state.
h = High voltage level one set-up time prior to the LOW-to-HIGH clock transition.
L = LOW voltage level steady state.
l = LOW voltage level one set-up time prior to the LOW-to-HIGH clock transition.
X = Don't care.
↑ = LOW-to-HIGH clock transition.

*Note*: (1) Both outputs will be HIGH while both $\bar{S}_D$ and $\bar{R}_D$ are LOW, but the output states are unpredictable if $\bar{S}_D$ and $\bar{R}_D$ go HIGH simultaneously.

(*continues*)

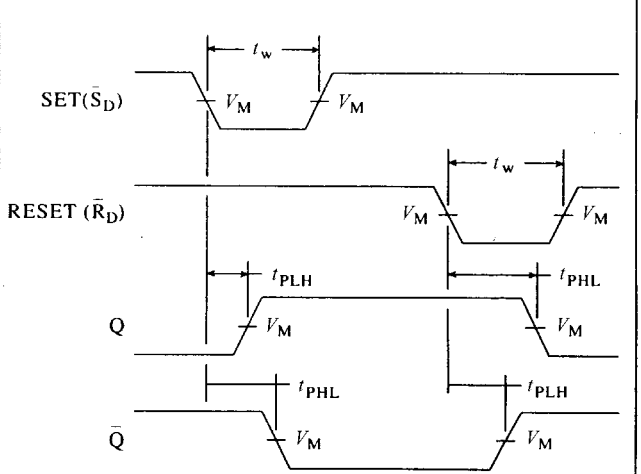
7474 (Continued)

AC WAVEFORMS




$V_M$ = 1.3 V for 74LS; $V_M$ = 1.5 V for all other TTL families. The shaded areas indicate when the input is permitted to change for predictable output performance.

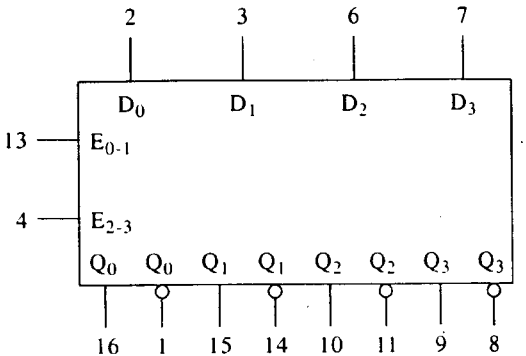
Waveform 1. Clock To Output Delays, Data Set-up And Hold Times, Clock Pulse Width




$V_M$ = 1.3 V for 74LS; $V_M$ = 1.5 V for all other TTL families.

Waveform 2. Set And Reset To Output Delays, Set And Reset Pulse Widths

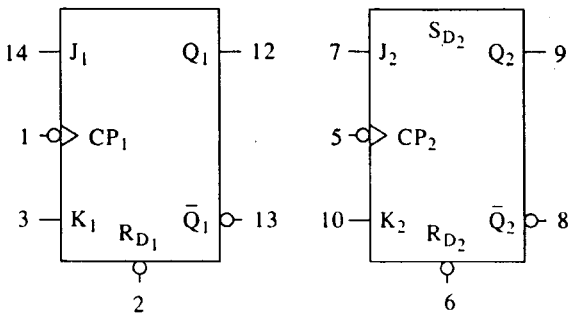
## 7475 A Latch IC



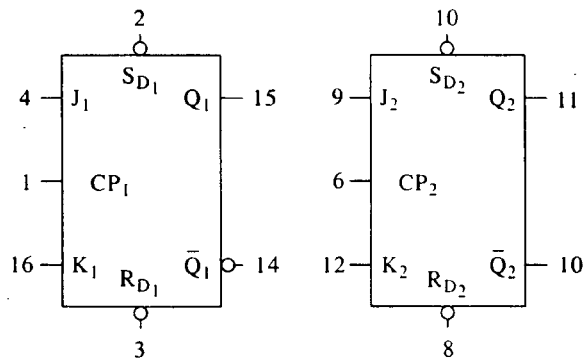

$V_{CC}$ = Pin 5
GND = Pin 12

## 7473 A Dual JK Flip-Flop IC

'LS73




'73



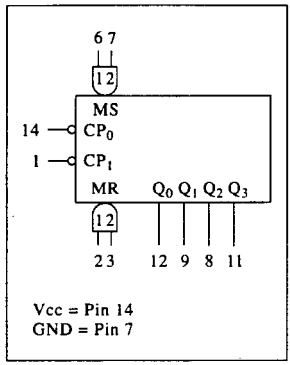
## 7490

The '90 is a 4-bit, ripple-type decade counter. The device consists of four master–slave flip-flops internally connected to provide a divide-by-two section and a divide-by-five section. Each section has a separate clock input to initiate state changes of the counter on the high-to-low clock transition. State changes of the Q outputs do not occur simultaneously because of internal ripple delays. Therefore, decoded output signals are subject to decoding spikes and should not be used for clocks or strobes.

A gated AND asynchronous master reset ($MR_1 \cdot MR_2$) is provided which overrides both clocks and resets (clears) all the flip-flops. Also provided is a gated AND asynchronous master set ($MS_1 \cdot MS_2$) which overrides the clocks and the MR inputs, setting the outputs to nine (HLLH).

Since the output from the divide-by-two section is not internally connected to the succeeding stages, the device may be operated in various counting modes. In a BCD (8421) counter the $\overline{CP}_1$ input must be externally *con*nected to the $Q_0$ output. The $CP_0$ input receives the incoming count producing a BCD count sequence. In a symmetrical bi-quinary divide-by-ten counter the $Q_3$ output must be connected externally to the $\overline{CP}_0$ input. The input count is then applied to the $CP_1$ input and a divide-by-ten square wave is obtained at output $Q_0$. To operate as a divide by two and a divide-by-five counter no external interconnections are required. The first flip-flop is used as a binary element for the divide-by-two function ($\overline{CP}_0$ as the input $Q_0$ as the output). The $\overline{CP}_1$ input is used to obtain a divide-by-five operation at the $Q_3$ output.

LOGIC SYMBOL




MODE SELECTION-
FUNCTION TABLE

| RESET/SET INPUTS | | | | OUTPUTS | | | |
|---|---|---|---|---|---|---|---|
| $MR_1$ | $MR_2$ | $MS_1$ | $MS_2$ | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
| H | H | L | X | L | L | L | L |
| H | H | X | L | L | L | L | L |
| X | X | H | H | H | L | L | H |
| L | X | L | X | | COUNT | | |
| X | L | X | L | | COUNT | | |
| L | X | X | L | | COUNT | | |
| H | L | L | X | | COUNT | | |

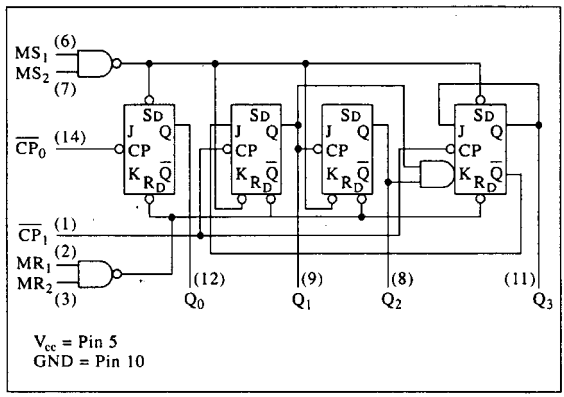
H = HIGH voltage level
L = LOW voltage level
X = Don't care

BCD COUNT SEQUENCE-
FUNCTION TABLE

| COUNT | OUTPUTS | | | |
|---|---|---|---|---|
| | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
| 0 | L | L | L | L |
| 1 | H | L | L | L |
| 2 | L | H | L | L |
| 3 | H | H | L | L |
| 4 | L | L | H | L |
| 5 | H | L | H | L |
| 6 | L | H | H | L |
| 7 | H | H | H | L |
| 8 | L | L | L | H |
| 9 | H | L | L | H |

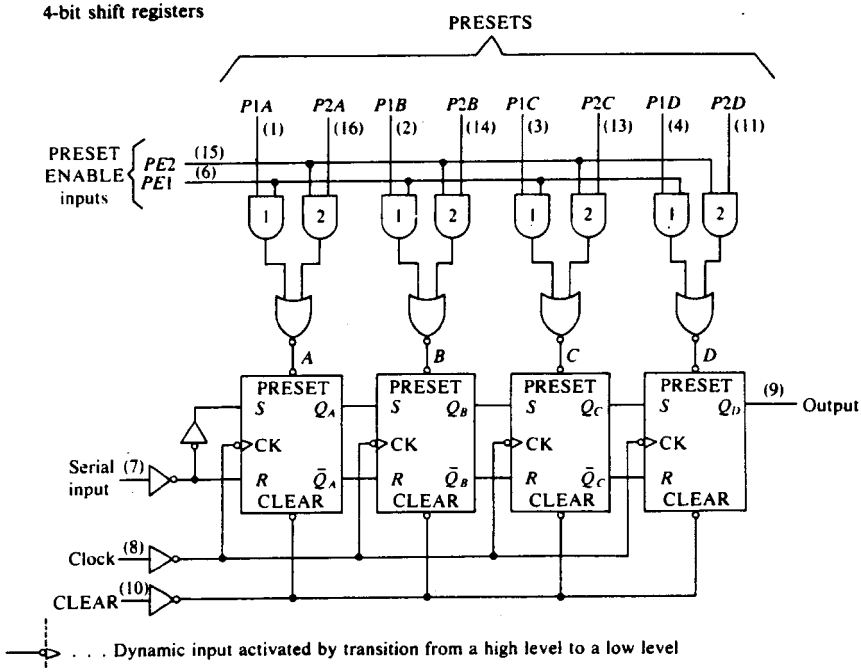
Note: Output $Q_0$ connected to input $\overline{CP}_1$

LOGIC SYMBOL




7490 (Continued)

**SN7494**
**4-bit shift registers**



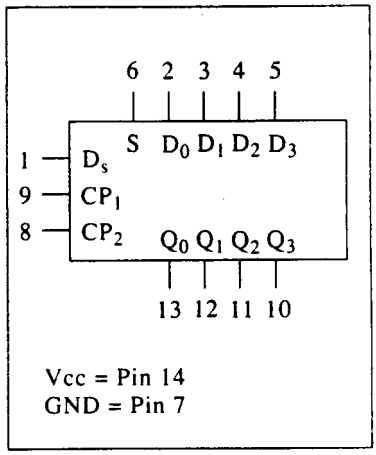
## 7495

The '95 is a 4-bit shift register with serial and parallel synchronous operating modes. It has serial data ($D_S$) and four parallel data ($D_0 - D_3$) inputs and four parallel outputs ($Q_0 - Q_3$). The serial or parallel mode of operation is controlled by a mode select input (S) and two clock inputs ($\overline{CP}_1$ and $\overline{CP}_2$). The serial (shift right) or parallel data transfers occur synchronously with the high-to-low transition of the selected clock input.

When the mode select input (S) is high, $\overline{CP}_2$ is enabled. A high-to-low transition on enabled $\overline{CP}_2$ loads parallel data from the $D_0 - D_1$ inputs into the register. When S is low, $CP_1$ is enabled. A high-to-low transition on enabled $\overline{CP}_1$ shifts the data from Serial input $D_S$ to $Q_0$ and transfers the data in $Q_0$ to $Q_1$, $Q_1$ to $Q_2$, and $Q_2$ to $Q_3$ respectively (shift right). Shift left is accomplished by externally connecting $Q_3$ to $D_2$, $Q_2$ to $D_1$, $Q_1$ to $D_0$, and operating the '95 in the parallel mode (S = high).

In normal operations the mode select (S) should change states only when both clock inputs are low. However, changing S from high-to-low while $\overline{CP}_2$ is low, or changing S from low-to-high while $\overline{CP}_1$ is low will not cause any changes on the register outputs

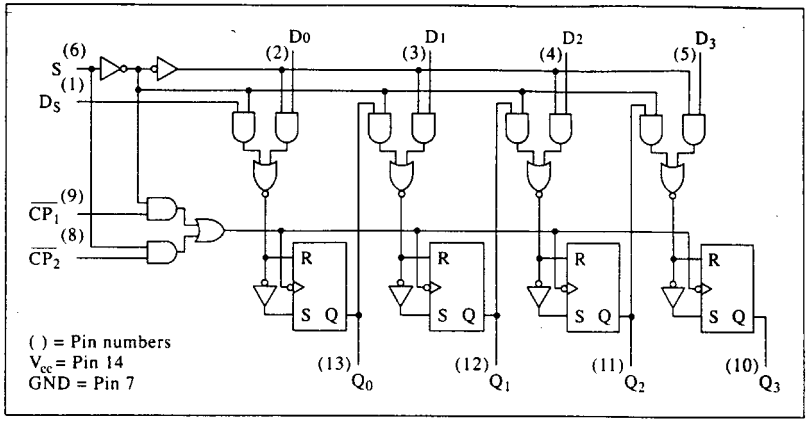
LOGIC SYMBOL




7495 4-Bit Shift Register

## 7495 (Continued)

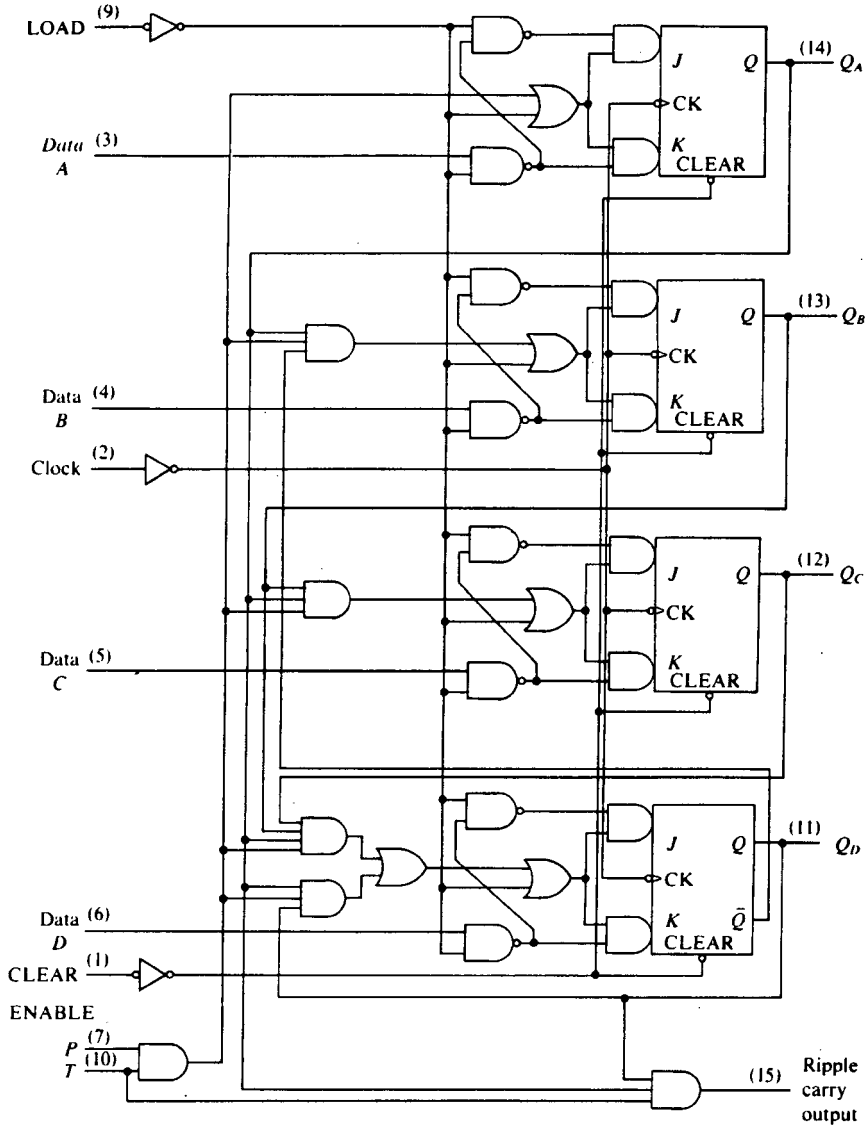
LOGIC DIAGRAM




FUNCTION TABLE

| OPERATING MODE | INPUTS | | | | | OUTPUTS | | | |
|---|---|---|---|---|---|---|---|---|---|
| | S | $CP_1$ | $CP_2$ | $D_S$ | $D_N$ | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
| Parallel load | H<br>H | X<br>X | ↓<br>↓ | X<br>X | l<br>h | L<br>H | L<br>H | L<br>H | L<br>H |
| Shift right | L<br>L | ↓<br>↓ | X<br>X | l<br>h | X<br>X | L<br>H | $q_0$<br>$q_0$ | $q_1$<br>$q_1$ | $q_2$<br>$q_2$ |
| Mode change | ↑<br>↑<br>↓<br>↓ | L<br>H<br>X<br>X | X<br>X<br>L<br>H | X<br>X<br>X<br>X | X<br>X<br>X<br>X | no change<br>undetermined<br>no change<br>undetermined | | | |

H = high voltage level steady state.
h = high voltage level one set-up time prior to the high-to-low clock transition.
L = low voltage level steady state.
l = low voltage level one set-up time prior to the high-to-low clock transition.
q = lower case letters indicate the state of the referenced output one set-up time prior to the high-to-low clock transition.
X = Don't care.
↓ = high-to-low transition of clock or mode select.
↑ = low-to-high transition of mode select.




74160 Synchronous Decade Counter

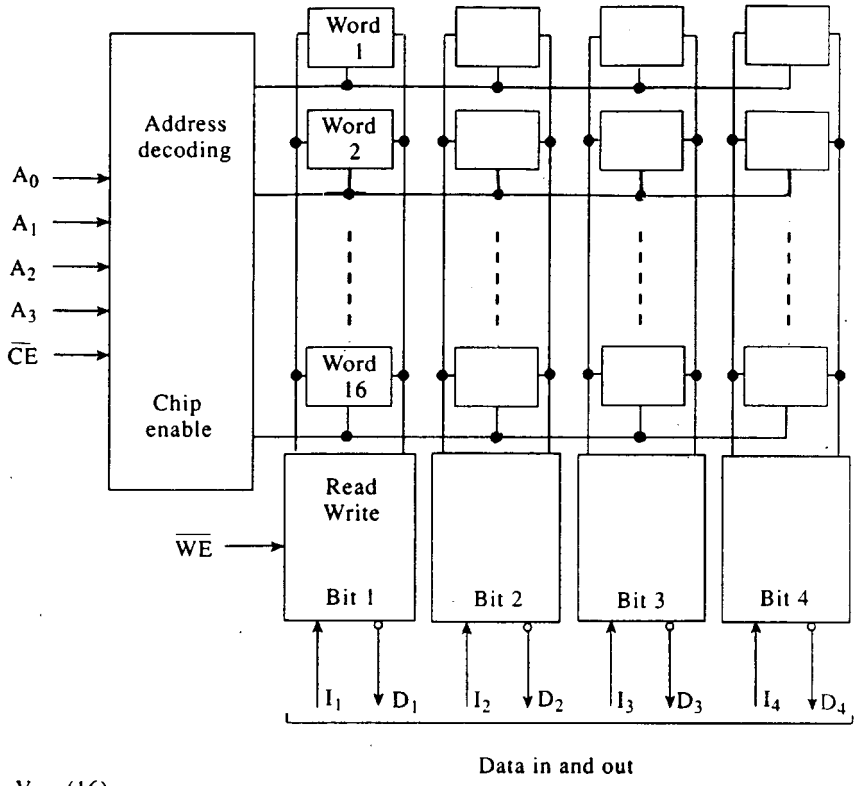
## C.3 MEMORY ICs

### Signetics 74S189

This 64-bit RAM is organized as 16 words of four bits each. There are four address lines ($A_0 - A_3$), four data input lines ($I_1 - I_4$), and four data output lines ($D_1 - D_4$). Note that the data output lines are active-low. Therefore, the output will be the complement of the data in the selected word. If the low-active chip enable ($\overline{\text{CE}}$) is high, the data outputs assume the high impedance state. When the write enable ($\overline{\text{WE}}$) is low, the data from the input lines are written into the addressed location. When the WE is high, the data are read from the addressed location. The operation of the IC is summarized in the truth table.

The timing characteristics of this IC are also shown in the figure. During a read operation, the data appear on the output $T_{\text{AA}}$ ns after the address is stable on the address inputs. $T_{\text{CE}}$ indicates the time required for the output data to be stable after $\overline{\text{CE}}$ is activated, and $T_{\text{CD}}$ is the chip disable time. During a write, once data on the input lines and address lines are stabilized, the $\overline{\text{CE}}$ is first activated, and $\overline{\text{WE}}$ is activated after a minimum data setup time of $T_{\text{WSC}}$. The $\overline{\text{WE}}$ must be active for at least $T_{\text{WP}}$ ns for a successful write operation.

**BLOCK DIAGRAM**



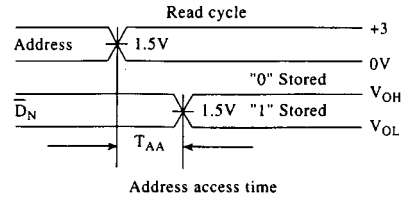

$V_{cc}$=(16)
GND=(8)

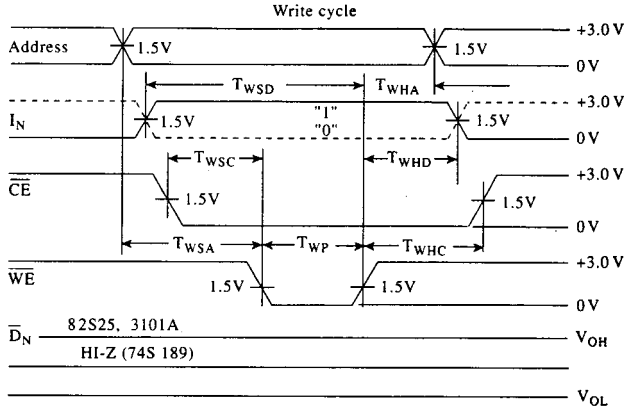
745189

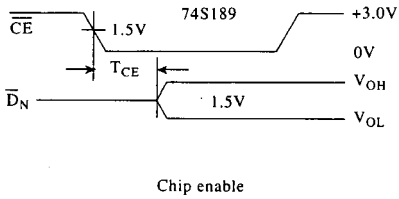
TRUTH TABLE

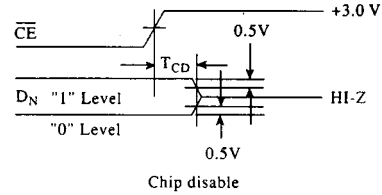
| Mode | $\overline{CE}$ | $\overline{WE}$ | $D_{IN}$ | 74S189 |
|---|---|---|---|---|
| Read | 0 | 1 | X | Stored data |
| Write "0" | 0 | 0 | 0 | Hi-Z |
| Write "1" | 0 | 0 | 1 | Hi-Z |
| Disable | 1 | X | X | Hi-Z |

TIMING DIAGRAMS




74S189 (Continued)

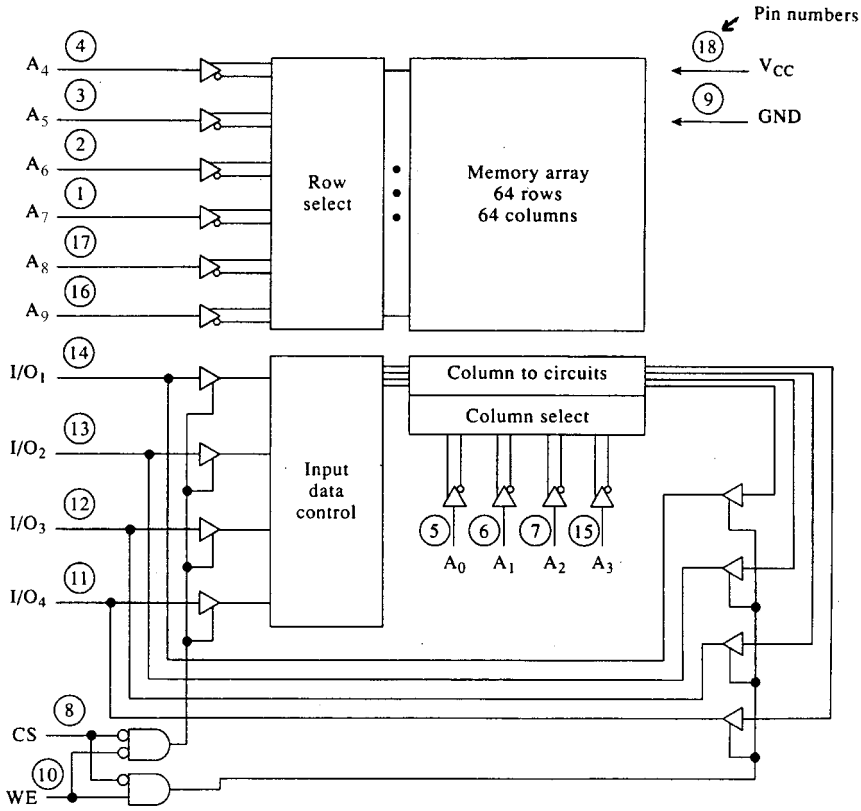
## Intel 2114

This is a 4096-bit static RAM organized as 1024 by 4. Internally, the memory cells are organized in a 64-by-64 matrix. There are 10 address lines ($A_0 - A_9$). Address bits $A_3 - A_8$ select one of the 64 rows. A four-bit portion of the selected row is selected by address bits $A_0$, $A_1$, $A_2$, and $A_9$. There is an active-low chip select ($\overline{CS}$). When the $\overline{WE}$ is low, the IC is put into a write

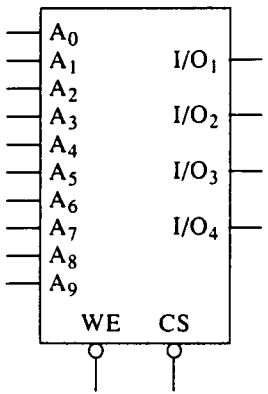
mode; otherwise, the IC will be in a read mode (if the $\overline{CS}$ is low). When the $\overline{CS}$ is high, the outputs assume the high impedance state. The common data I/O lines thus are controlled by $\overline{CS}$ and $\overline{WE}$.

This device uses HMOS II, a high-performance MOS technology, and is directly TTL compatible in all respects: inputs, outputs, and single 5V power supply. This device is available in five versions. The maximum access time ranges from 100 to 250 ns depending on the version. The maximum current consumption ranges from 40 to 70 mA.

## BLOCK DIAGRAM



LOGIC SYMBOL




PIN NAMES

| | | |
|---|---|---|
| $A_0$–$A_9$ | Address inputs | $V_{CC}$ power (+5V) |
| $\overline{WE}$ | Write enable | GND ground |
| $\overline{CS}$ | Chip select | |
| $I/O_1$–$I/O_4$ | Data input/output | |

Intel 2114 (Continued)

## Texas Instruments TMS4116

This is a 16K × 1 dynamic NMOS RAM. It has a data input ($D$), a data output ($Q$), and a read/write control ($\overline{W}$) input. To decode 16K, 14 address lines are required. The IC provides only seven address lines ($A_0$–$A_6$). The 14 address bits are multiplexed onto these seven address lines using row address select ($\overline{RAS}$) and column addres select ($\overline{CAS}$) inputs. Although this multiplexing decreases the speed of the RAM operation, it minimizes the number of pins on the IC. Three power supplies (12 V, + 5 V, and −5 V) are required.

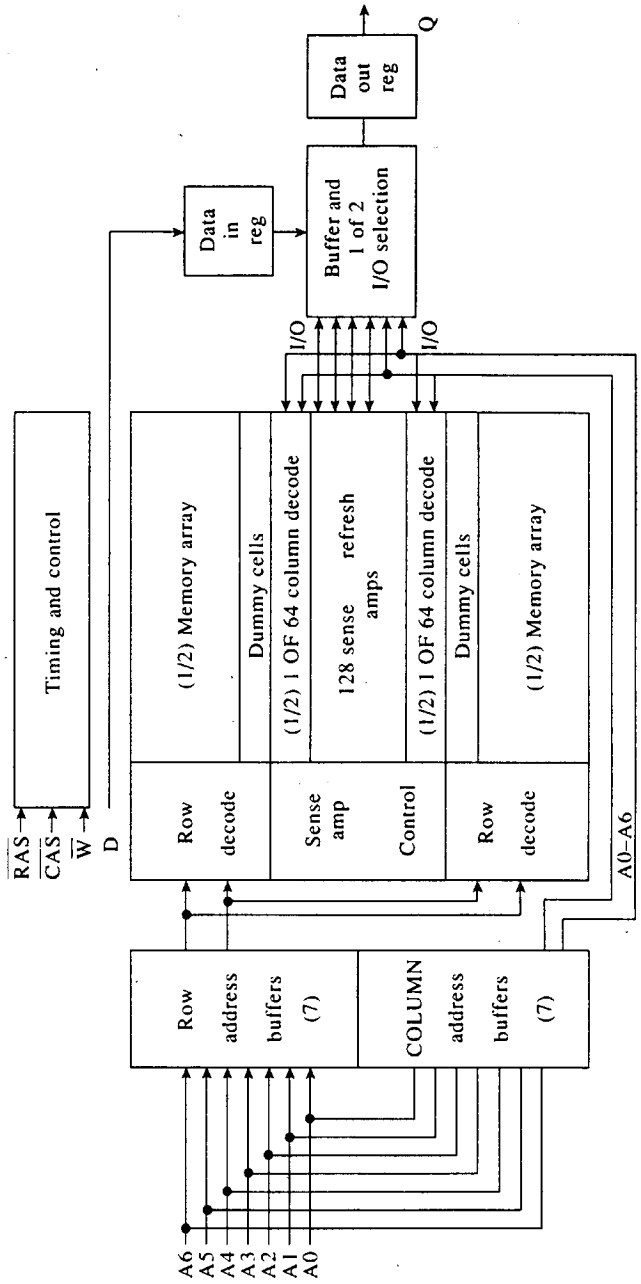
The operation of the IC is illustrated in (a). The memory cells are organized in a 128 × 128 array. The seven low-order address bits select a row. During a read, the data from the selected row are transferred to 128 sense/refresh amplifiers. The seven high-order address bits then select one of the 128 sense amplifiers and connect it to the data output line. At the same time, the data on the sense amplifiers are refreshed (i.e., the capacitors are charged) and rewritten to the proper row in the memory array. During a write, the data in the sense amplifiers are changed to new data values just before the rewrite operation. Thus, at each read or write cycle, a row of the memory is refreshed.

The timing diagrams for read, refresh, and write cycles are shown in (b), (c), and (d). When $\overline{RAS}$ and $\overline{CAS}$ are both high, $Q$ will be at a high impedance state. To begin a cycle, the seven low-order address bits are first placed on the address lines and the $\overline{RAS}$ is driven low. The 4116 then latches the row address. The high-order seven bits of the address are then placed on address lines and the $\overline{CAS}$ is driven low, thus selecting the required bit. For a write cycle, $D$ must have valid data when the $\overline{CAS}$ goes low and the $\overline{W}$ should go low just after $\overline{CAS}$ does. For a read cycle, $Q$ will be valid after $t_{a(c)}$ from the start of $\overline{CAS}$ and will remain valid until $\overline{RAS}$ and $\overline{CAS}$ go high.

The data in the memory must be refreshed every 2 ms. Refreshing is done by reading a row of data into the sense amplifiers and rewriting it. Only $\overline{RAS}$ is required to peform the refresh cycle. A seven-bit counter can be used to refresh all the rows in the memory. The counter counts up every 2 ms. The seven-bit output of the counter becomes the row address at each cycle.

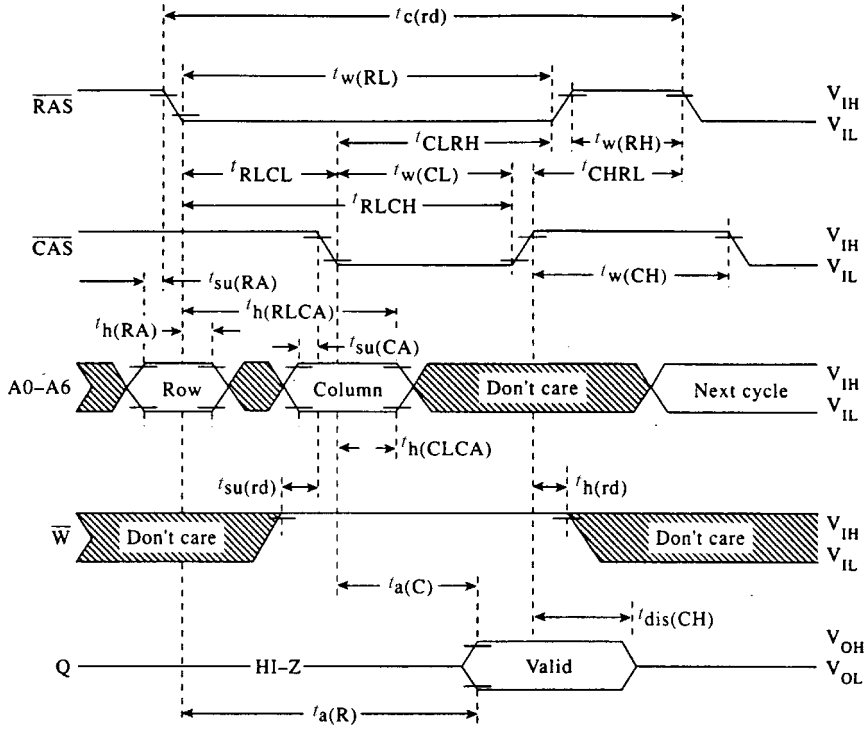
Two modes of refresh are possible: *burst* and *periodic*. In a burst mode, all rows are refreshed every 2 ms. Thus, if each refresh cycle takes about 450 ns, in a burst refresh mode, the first (128 × 450 = 57600 ns) 57.6 $\mu$s will be consumed by the refresh and 1942.4 $\mu$s will be available for the read and write. In a periodic mode, there will be one refresh cycle every (2/128 =) 15.626 $\mu$s. The first 450 ns at each 15.625 $\mu$s interval will be taken for the refresh.

Several dynamic memory controllers are available off-the-shelf. These controllers generate appropriate signals to refresh the dynamic memory module in the system. One such controller is described next.



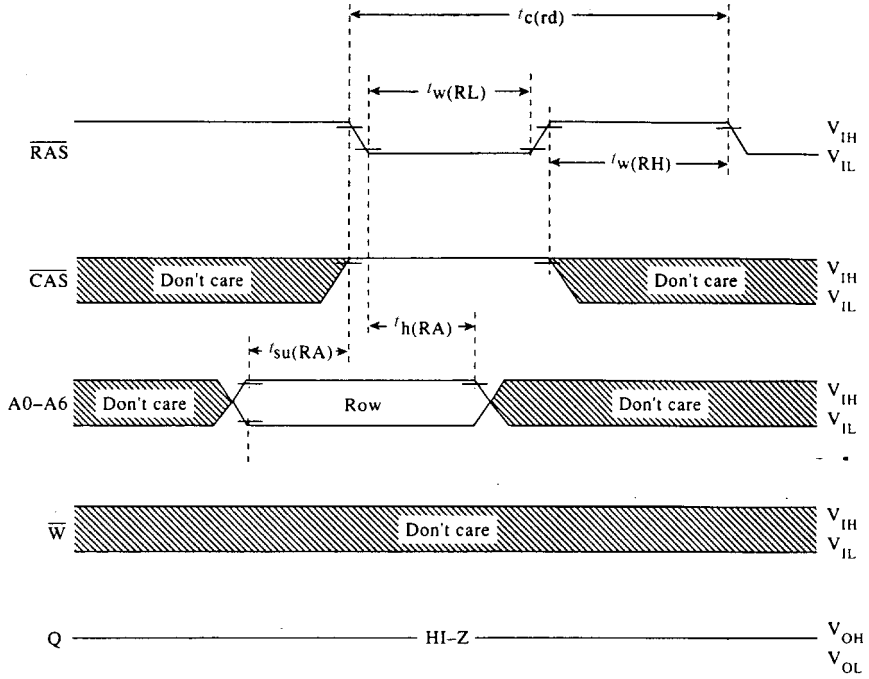

(a) Functional block diagram

A 16384-bit dynamic RAM (TMS4116) (Courtesy of Texas Instruments, Inc.)



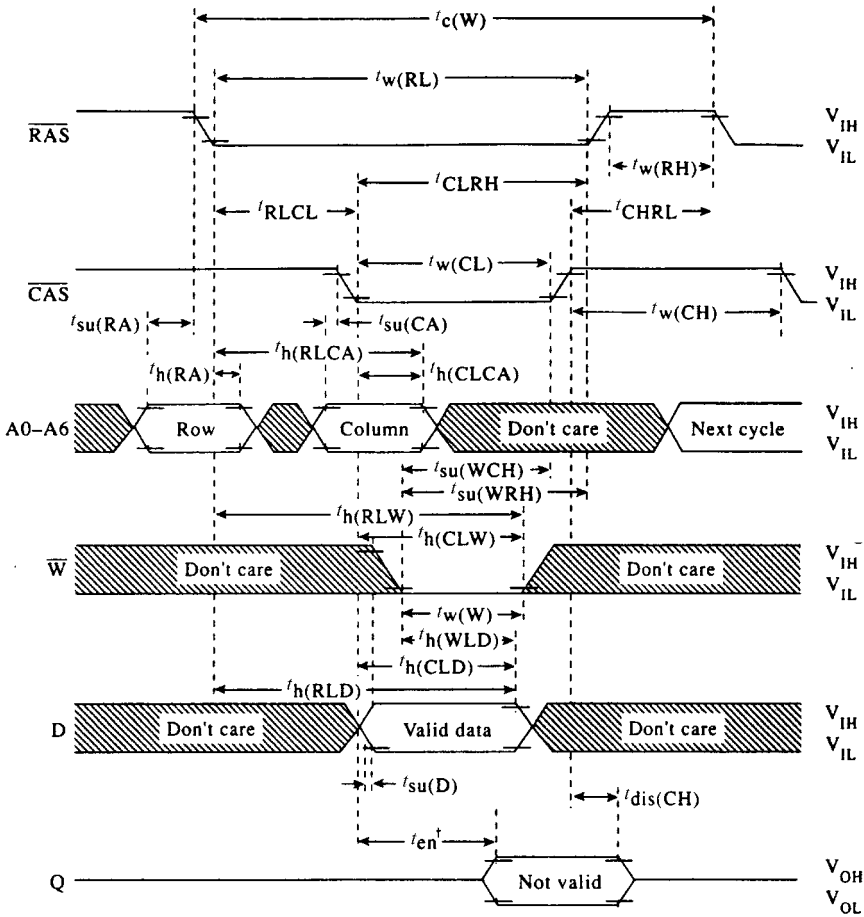

(b) Read cycle timing

TMS4116 (Continued)




(c) $\overline{RAS}$-only refresh timing

TMS4116 (Continued)




(d) Write cycle timing

TMS4116 (Continued)

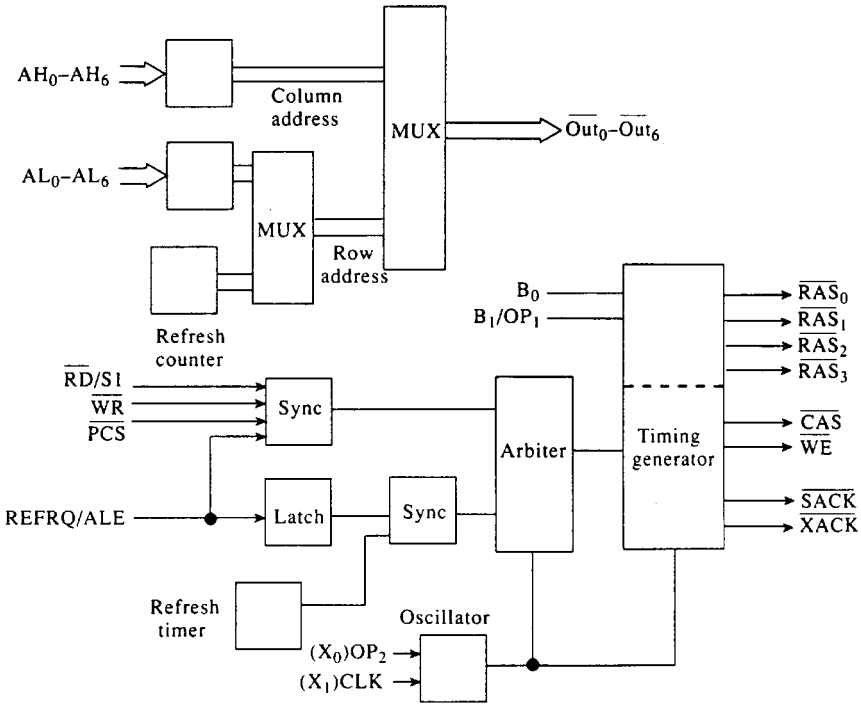
## INTEL 8202A Dynamic RAM Controller

This device provides all the signals needed to control a 64K dynamic RAM of the TMS4116 type. It performs address multiplexing and generates $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals. It contains a refresh counter and timer. The device has several other features that make it useful in building microcomputer systems. In what follows, we will concentrate on the basic features needed to

control a dynamic RAM. The reader should refer to the manufacturers' manuals for further details.

Functions of all the pins on the device are shown in (b). The outputs ($\overline{OUT}_0$–$\overline{OUT}_6$) are functions of either the 14-bit address inputs ($AL_0$–$AL_6$ and $AH_0$–$AH_6$) or the refresh counter outputs. The outputs of the 8202A are directly connected to the address inputs of the 4116. The $\overline{WE}$ is connected to the $\overline{W}$ of the memory. There is a chip select input ($\overline{PCS}$) and a clock (CLK) input. In addition to the $\overline{CAS}$, four $\overline{RAS}$ signals are generated by the device. These multiple $\overline{RAS}$ signals are useful in selecting a bank of the memory when larger memory systems are built using dynamic memory ICs. Signals such as $\overline{RD}$, $\overline{WR}$, $\overline{XACK}$, and $\overline{SACK}$ are compatible with the control signals produced by microprocessors (such as Intel 8088).

Dynamic RAM controller (Intel 8202A). (Reprinted by permission of Intel Corp. Copyright 1983. All mnemonics copyright Intel Corp. 1986.)




8202A BLOCK DIAGRAM

(a)

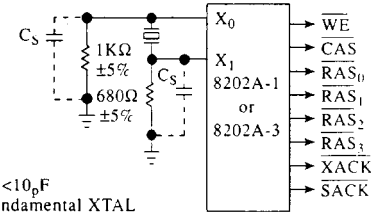
INTEL 8202 A

**Pin descriptions**

| Symbol | Pin No. | Type | Name and function |
|---|---|---|---|
| $AL_0$<br>$AL_1$<br>$AL_2$<br>$AL_3$<br>$AL_4$<br>$AL_5$<br>$AL_6$ | 6<br>8<br>10<br>12<br>14<br>16<br>18 | I<br>I<br>I<br>I<br>I<br>I<br>I | **Address low:** CPU address inputs used to generate memory row address. |
| $AH_0$<br>$AH_1$<br>$AH_2$<br>$AH_3$<br>$AH_4$<br>$AH_5$<br>$AH_6$ | 5<br>4<br>3<br>2<br>1<br>39<br>38 | I<br>I<br>I<br>I<br>I<br>I<br>I | **Address high:** CPU address inputs used to generate memory column address. |
| BO<br>$B_1/OP_1$ | 24<br>25 | I<br>I | **Bank select inputs:** Used to gate the appropriate $RAS_0$- $RAS_3$ output for a memory cycle. $B_1/OP_1$ option used to select the Advanced Read Mode. |
| $\overline{PCS}$ | 33 | I | **Protected chip select:** Used to enable the memory read and write inputs. Once a cycle is started, it will not abort even if $\overline{PCS}$ goes inactive before cycle completion. |
| $\overline{WR}$ | 31 | I | **Memory write request.** |
| $\overline{RD}$/S1 | 32 | I | **Memory read request:** S1 function used in Advanced Read mode selected by $OP_1$ (pin 25). |
| REFRQ/<br>ALE | 34 | I | **External refresh request:** ALE function used in Advanced Read mode, selected by $OP_1$ (pin 25). |
| $\overline{OUT}_0$<br>$\overline{OUT}_1$<br>$\overline{OUT}_2$<br>$\overline{OUT}_3$<br>$\overline{OUT}_4$<br>$\overline{OUT}_5$<br>$\overline{OUT}_6$ | 7<br>9<br>11<br>13<br>15<br>17<br>19 | O<br>O<br>O<br>O<br>O<br>O<br>O | **Output of the multiplexer:** These outputs are designed to drive the addresses of the Dynamic RAM array. (Note that the pins do not $\overline{OUT}_{0-6}$ require inverters or drivers for proper operation.) |
| $\overline{WE}$ | 28 | O | **Write enable:** Drives the Write Enable inputs of the Dynamic RAM array. |
| $\overline{CAS}$ | 27 | O | **Column address strobe:** This output is used to latch the column address into the dynamic RAM array. |

| Symbol | Pin No. | Type | Name and function |
|---|---|---|---|
| $\overline{RAS}_0$<br>$\overline{RAS}_1$<br>$\overline{RAS}_2$<br>$\overline{RAS}_3$ | 21<br>22<br>23<br>26 | O<br>O<br>O<br>O | **Row address strobe:** Used to latch the Row Address into the bank of dynamic RAMs, selected by the 8202A Bank Select pins ($B_0$, $B_1/OP_1$). |
| $\overline{XACK}$ | 29 | O | **Transfer acknowledge:** This output is a strobe indicating valid data during a read cycle or data written during a write cycle. $\overline{XACK}$ can be used to latch valid data from the RAM array. |
| $\overline{SACK}$ | 30 | O | **System acknowledge:** This output indicates the beginning of a memory access cycle. It can be used as an advanced transfer acknowledge to eliminate wait states. (Note: If a memory access request is made during a refresh cycle, $\overline{SACK}$ is delayed until $\overline{XACK}$ in the memory access cycle). |
| $(X_0)$ $OP_2$<br>$(X_1)$ CLK | 36<br>37 | I/O<br>I/O | **Oscillator inputs:** These inputs are designed for a quartz crystal to control the frequency of the oscillator. If $X_0/OP_2$ is connected to a 1 KΩ resistor pulled to +12 V then $X_1$/CLK becomes a TTL input for an external clock. |
| N.C. | 35 | | Reserved for future use. |
| $V_{CC}$ | 40 | | **Power supply:** +5 V. |
| GND | 20 | | **Ground.** |

**NOTE:** Crystal mode for the 8202A-1 or 8202A-3 only.




**Crystal Operation for the 8202A-1 and the 8202A-3**

(b)

INTEL 8202A (Continued)

## REFERENCES

*Bipolar Memory Data Manual*, Sunnyvale, CA: Signetics, 1987.

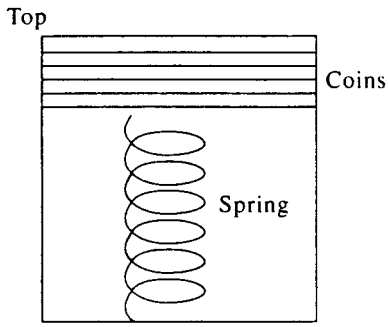
*Bipolar Microcomputer Components Data Book*, Dallas, TX: Texas Instruments, 1987.

*FAST TTL Logic Series Data Handbook*, Sunnyvale, CA: Phillips Semiconductors, 1992.

*TTL Data Manual*, Sunnyvale, CA: Signetics, 1987.

# Appendix D
## Stack Implementation

A last in/first-out (LIFO) stack is a versatile structure useful in a variety of operations in a computer system. It is used for address and data manipulation, return address storage and parameter passing during subroutine call and return, and arithmetic operations in an ALU. It is a set of storage locations or registers organized in a LIFO manner. A coin box (shown below) is the most popular example of a LIFO stack. Coins are inserted and retrieved from the same end (top) of the coin box. PUSHing a coin moves the stack of coins down one level, the new coin occupying the top level (TL). POPing the coin box retrieves the coin on the top level. The second level (SL) coin becomes the new top level after the POP.




In a LIFO stack (or simply "stack"):

| | |
|---|---|
| PUSH implies | all the levels move down by one: TL ← data, and |
| POP implies | pops out ← TL, TL ← SL; all other levels move up. |

Two popular implementations of the stack are:

1. RAM-based implementation.
2. Shift-register-based implementation.

In a RAM-based implementation, a special register called a *stack pointer* (SP) is used to hold the address of the top level of the stack. The stack is built in a reserved area in the memory. The PUSH operation then corresponds to

SP ← SP + 1.
MBR ← data.
MAR ← SP.
WRITE MEMORY.

The POP operation corresponds to

MAR ← SP.
READ MEMORY.
Output ← MBR. "Output" is the destination for data from
SP ← SP – 1. the top level.

In this implementation, the stack grows toward higher address memory locations as items are PUSHed into it. There is not actual movement of data between the levels during PUSH and POP operations.

Figure D.1 portrays the shift-register-based implementation of an *n*-level stack. Each stack level can hold an *m*-bit datum. Data are PUSHed into stack by using a SHIFT RIGHT signal and POPed out of stack by using a SHIFT LEFT signal. There is movement of data betwen levels in this implementation.

Shift-register-based implementations are faster than RAM-based stacks because no memory access is needed. RAM-based implementations are popular because no additional hardware is needed (other than SP) to implement a stack. Instructions to PUSH and POP registers and memory locations must be added to the instruction set once a stack is included in the design.

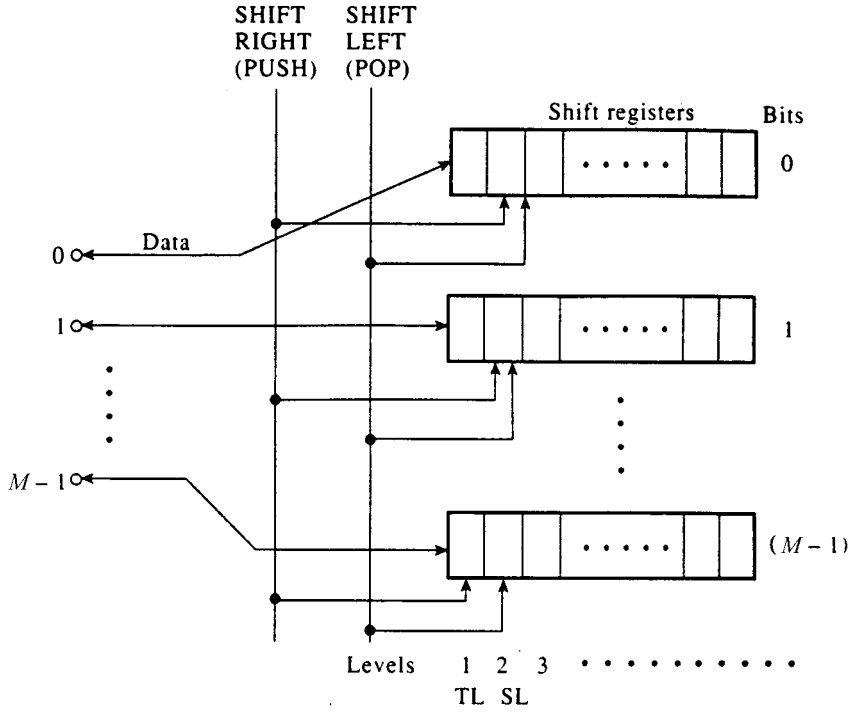



**Figure D.1** A shift-register-based stack

# Index


A simple computer (ASC), 181
  addressing modes, 191
  assembler, 203–210
  bus structure, 221–223
  console, 249–257
  control unit, 232–267
  data format, 184
  data, instruction, address flow, 217–221
  hardware components, 183
  input/output, 230–232
  instruction execution, 216
  instruction format, 184–186
  instruction set, 186–191
  program execution, 216–217
AND-OR circuits, 29–30
Absolute address, 187, 188, 198
Access time, 154
Accumulator register (ACC), 182
Adders, 36–40
  carry look ahead, 525
  full, 36, 37–39
  half, 36–38
  ripple-carry, 37
Address extension, 447
Address mapping function, 433
Addressing modes, 185, 379–384
  base-register, 381
  DEC PDP-11 of, 383
  DEC VAX of, 384
  direct, 185, 191, 196
  IBM 370 of, 384
  INTEL 8080 of, 382
  INTEL 8086 of, 382
  immediate, 195, 379
  implied, 383
  indexed, 185, 192–194
  indexed-indirect (pre-indexed indirect), 194, 195
  indirect, 185, 194, 195
  indirect-indexed (post-indexed indirect), 194, 195
  literal, 196
  M6502 of, 382
  MC6800 of, 382
  paged, 379
  relative, 381
Address modifiers, 380
Address translation, 433
Analog-digital converters (A/D), 312
Architectures
  classes of, 572
  RISC, 484
  dataflow, 617
  design Vs, 8
  distributed processing system, 618
  network, 618
Arithmetic, 635–645
  binary, 635
  decimal, 535
  hexadecimal, 642
  octal, 640
Arithmetic and logic unit (ALU), 523
Arithmetic shift, 524

Assembler, 417
Associative memory, 144, 147–149, 167

Backward daisy chain, 295
Banking, 426
Bank switching, 449
Base register, 381
Binary system, 626
Bipolar logic families, 57
  ECL, 57
  TTL, 57
Bit-parallel, 523
Bit-serial, 523
Boolean algebra (switching algebra), 21–22
Branch history, 479
Branch prediction, 479
Bus
  arbitrator, 300
  architecture, 299
  control, 300
  standards, 322
  transfer, 124

CPU-I/O device handshake, 281
CRAY
  X-MP, 555
  T3E, 592
Cache memory, 431
Carry, 11
Carry look ahead adder (CLA), 525
Channels, 300
Characteristic tables, 83
Class code method,
Code converters, 40
Codes, 651
  alphanumeric, 653
  BCD, 651
  excess-3, 652
  two-out-of-five, 653
Coincident decoding, 158
Combinational circuits, 11
  analysis of, 23–28
  synthesis of, 29–35
Compact disk ROMs (CDROM), 156
Compaq SP700, 409
Compatible I/O bus structure, 299
Complex instruction set computers (CISC), 365
Computer
  architectures, 7
  evolution, 5–8
  generations, 6
  levels of, 8
  system organization, 2–4
Computer networks, 618
Connection machine
  CM-1, 580
  CM-5, 589
Content-addressable memory (CAM) (*see* Associative memory)
Control data 6600, 6, 353, 552
Control ROM (CROM), 257
Cray,
  series, 555
  T3E, 592
Cycle stealing, 298
Cycle time, 154

DEC
  PDP-11 system, 348, 388
  VAX-11/750, 492
Daisy chain, 292
Data conversion, 279, 284
Data transfer rate, 154
DeMorgan's Law, 22
Decoders, 40
Delayed branching, 479
Demultiplexers, 45
Device
  controller, 283
  interface, 277
  priority, 292
Digital to analog converters (D/A), 312
Diminished radix complement, 649
Direct mapping, 435
Direct memory access, 295

Direct-access memory (DAM), 144, 152, 171
  erasable disks, 171
  floppy disks, 171
  hard disks, 171
  magnetic disks, 171
  optical disks, 171
  WORM, 172
Disk coupled system, 303
Division, 528
Dynamic memory, 160
Dynamic RAM, 160

EDSAC, 6
EDVAC, 6
ENIAC, 6
Edge-triggered flip-flops, 89
Encoders, 42
Erasable disks, 172
Error detection and correction, 284
Even parity, 286
Excitation tables, 83
Execute phase, 221

FIFO, 150
Fetch phase, 217
Flip-flops, 76
  characteristic and excitation tables, 83
  delay (D), 81
  edge-triggered, 89
  ICs, 91
  JK, 81
  master-slave, 87
  set-reset (SR), 76–80
  T (toggle), 83
  timing, characteristics of, 85
Floating-point binary, 362
Forward daisy chain, 293
Front end processor, 302
Fully associative mapping, 435

HDL, 128
Hardwired control unit (HCU), 481
Hazards, 67
Hexadecimal system, 629
Hit ratio, 441
Horizontal microinstruction, 483

I/O
  address space, 276
  devices, 310
  function, 279
  modes of, 273
  processor, 302–305
ILLIAC-IV, 578
INTEL
  2114, 689
  21285, 338
  8080, 387, 489
  8086, 387
  8089, 335
  8202A, 706
  pentium II, 392
Immediate addressing, 379
Implied (implicit) addressing, 382
Index register, 182
Indexed addressing, 192
Indirect addressing, 193
Input/output (*see* I/O)
Instruction
  buffer, 430, 480
  cache, 480
  classification, 365
  cycle, 216
  formats, 371
  length, 366
  opcode selection, 369
  register (IR), 182
  set, 364
  stream, 572
Integrated circuits (ICs), 47–64, 679
  7400, 680
  7401, 680
  7402, 680
  7404, 680
  7406, 680
  7451, 680
  7474, 688
  7473, 690

[Integrated circuits (ICs)]
7483, 682
7490, 691
7494, 693
7495, 694
74150, 683
74155, 685
74181, 539
characteristics of, 52
designing with, 61
hazards, 67
Intel 2114, 699
Intel 8202, 706
loading and timing, 64
Signetics 74189, 697
Texas Instruments TMS 4116, 702
technologies, 47–48
types of, 47–48
Integrated dynamic RAMs, 163
Interleaving, 428
Interrupt mode I/O, 273, 283
Interrupt service routine, 286
Interrupt structures, 295
multipriority vectored, 295
single priority, 295
polled, 295
vectored, 295
Interrupts, 286
conditions for, 286
mechanism for ASC, 287
multiple interrupts, 290
polling, 290
types of interrupt structures, 294
vectored interrupts, 291
Isolated I/O mode, 276

JK flip-flop, 81

Karnaugh map (KMap), 659

LIFO SAM, 150
LRU, 437
Linear decoding, 158
Literal addressing mode, 200
Loader program, 209
Loading analysis, 64
Location counter, 202

Machine language programming, 197
Macroinstructions, 365
Mapping function, 433
fully associative, 435
Master-slave flip-flop, 87
Memory
bank, 426
CAM, 144, 146
DAM, 144, 152
Design Using ICs, 173
Devices and Organization, 156–173
ICs,
Hierarchy, 155
PLA, 164
RAM, 144
ROM, 144, 146
RWM, 145
SAM, 144–150
System Parameters, 152
Types of, 144
Memory-mapped I/O, 276
Microprogrammed control unit (MCU), 481
MIMD machine, 572, 575
Minimization of Boolean functions, 657
K maps, 659
QM method, 673
Venn diagrams, 657
MIPS R10000, 498
MISD machine, 572, 574
Miss ratio, 441
Mnemonic, 197
MOS logic families, 48
CMOS, 48
NMOS, 48
PMOS, 48
MOS technology 6502, 370
Motorola
6800, 387

[Motorola]
  68000, 314
  68020, 465
  68881, 549
  88100, 494
  88200, 495
Multibus
  I, 322
  II, 325
Multiple-bit shift, 525
Multiple-match resolver, 147
Multiplexer channel, 302
Multiplexers, 44
  block, 302
  character, 302
Multiplication, 527, 637, 640, 643
Multiport memories, 429

NAND-NAND circuits, 30
NOR-NOR circuits, 30
Negative logic, 48
Nonrestoring division algorithm, 531

OR-AND circuits, 30
Octal system, 628
Odd parity, 285
One address instructions, 188
One-address machine, 367
Opcode table, 205
Opcode, 186

PROM, 164
  electrically alterable (EAROM), 164
  erasable, 164
  programmer, 164
Page
  fault, 444
  replacement policy, 437, 443
  size, 444
  table, 448
Parity bits, 284
Peripheral processors, 302
Physical address, 442
Pipeline, 536
Polling, 290
Positive logic, 48
Postindexed-indirect, 194
Preindexed-indirect, 194
Primary memory, 154
Printers, 311
Program counter (PC), 182
Program loading, 209
Program locality principle, 432
Programmable logic array (PLA), 164
Programmable logic sequencers, 166
Programmed I/O, 273
Pseudo-instruction, 199

Quine-Mckluskey procedure, 673

RS-232 interface, 308
Random access memory (RAM), 144
  devices, 157
  dynamic memory, 160
  read/write memory, 145
  static RAM, 157
Read only memory (ROM), 146
  code converter with, 165
  mask-programmed, 163
  programmable logic arrays (PLA), 164
  user-programmed (PROM), 163
  Read/write memory (RWM), 145
  Reduced instruction set computers (RISC), 365, 484
Register
  bus transfer, 124
  CLEAR and LOAD with, 110
  Clearing of, 109
  HDL constructs, 130
  HDL operators, 128
  JK flip-flops used, 112
  Loading of, 109
  point to point transfer, 124
  transfer languages, 128
  transfer logic of, 119
  transfer, 120
Relative addressing, 381
Replacement algorithm, 433, 437

Reserved opcode method, 369
Restoring division algorithm, 531

Selector channel, 302
Signetics 74S189, 697
SIMD machine, 572, 573
SISD machine, 572
Scanner, 312
Secondary memory devices, 154
Secondary storage, 154
Segmentation systems, 443
Selector channel, 302
Semirandom access memory, 150
Sequence detector, 114
Sequential access memory (SM), 144, 168
Sequential circuit, 73–142
  asynchronous, 73
  design with ICs, 133
  synchronous, 73
Serial
  2's complementer, 118
  adder, 116
  I/O, 306
  transfer scheme, 120
Set associative
  cache, 435
  mapping, 435
Set-reset (SR) flip-flop, 76–80
Shared memory system, 304
Shift register, 113
Sign magnitude representation, 646
Signal levels, 49
Special outputs, 57
  buffers, 57
  tristate, 59
  wired-or, 58
Stack
  based ALU, 532
  implementation, 709
State
  diagram, 92
  table, 95, 96
  transitions, 93
Static RAM, 157
Symbol table, 203
Symbolic
  address, 198
  name (mnemonic), 197
Synchronous sequential circuits, 73–143
  analysis of, 92–102
  design of, 102–107

Terminals, 311
Texas Instruments
  MSP 430, 541
  TMS 4116, 702
Three-address machine, 366
Toggle (T) flip-flop, 83
Translation lookaside buffer (TLB), 447
Truth tables, 15
Two-address machine, 367
Two-pass assembler, 202

Universal asynchronous receiver transmitter (UART), 309

VME bus, 331
Vertical (packed) microinstruction, 484
Virtual address, 441
Virtual memory, 441
Von neumann machine, 6

Wider word fetch, 392
Write back, 440
Write once read many times (WORM), 156
Write through, 440

Zero address
  instructions, 187
  machine, 367